The SaM Synchronization Manager Distributed Object Oriented Programming FY92 Final Report

MTR 92B0000175
January 1993

Myra Jean Prelle
Thomas J. Brando

MITRE
Bedford, Massachusetts

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
January 1993

3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE
The SaM Synchronization Manager Distributed Object
Oriented Programming FY92 Final Report

5. FUNDING NUMBERS

6. AUTHOR(S)

Myra Jean Prelle
Thomas J. Brando

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

8. PERFORMING ORGANIZATION REPORT NUMBER

MTR 92B0000175

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

same as above

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

same as above

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public releasc; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT

We describe a multicomputer run-time executive called the Synchronization Manager (SaM). SaM makes it easier to develop, debug, and enhance multicomputer software because it automatically manages the synchronization required by an object-oriented program to produce the same results as a single-computer execution. We describe our experience with using SaM to run several application programs (image perspective transformation, neural network training, and multiple target tracking) written in an object-oriented extension of C, called CPM, on a Symult S2010 multicomputer. CPM is described and performance results are presented that suggest the viability of our approach.

14. SUBJECT TERMS

multicomputer, Synchronization Manager, object-oriented
extension of C

15. NUMBER OF PAGES
80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited

The SaM Synchronization Manager Distributed Object Oriented Programming FY92 Final Report

MTR 92B0000175

January 1993

Myra Jean Prelle
Thomas J. Brando

Contract Sponsor RL/C3AB
Contract No. F19628-89-C-0001
Project No. 7850
Dept. G040

MITRE
Bedford, Massachusetts

ABSTRACT

We describe a multicomputer run-time executive called the Synchronization Manager (SaM). SaM makes it easier to develop, debug, and enhance multicomputer software because it automatically manages the synchronization required by an object-oriented program to produce the same results as a single-computer execution. We describe our experience with using SaM to run several application programs (image perspective transformation, neural network training, and multiple target tracking) written in an object-oriented extension of C, called CPM, on a Symult S2010 multicomputer. CPM is described and performance results are presented that suggest the viability of our approach.

ACKNOWLEDGMENTS

The work described in this paper was performed by The MITRE Corporation under a project funded by the Air Force Electronic Systems Division. The project office is at the Rome Laboratory, Griffiss Air Force Base, Rome, New York. The project officer is Vaughn Combs.

In addition, we thank Ann Wollrath. She designed and implemented the CPM language. Her work with the neural network application led to her design and implementation of the assert method extension and the send to self optimization. We also wish to thank Lori Phillips for her thoughtful and tireless contributions to the work described in this paper.

TABLE OF CONTENTS

SECTION **PAGE**

LIST OF FIGURES

FIGURE **PAGE**

LIST OF TABLES

SECTION 1

INTRODUCTION

Performance requirements for modern military application programs dictate the need for more processing power than is available in conventional single-computer systems. Existing systems, or systems under development, that use parallel processors range from radar or other sensor systems, the Air Force/Army Joint Surveillance and Targeting System (Joint STARS) or the Navy's Sea Wolf Submarine Combat System (AN/BSY-2), to advanced Airspace Planning Functions (APF) for the FAA, weather and climate models for the FAA and NASA Earth Observation System (EOS), and the Automated Fingerprint Identification System (AFIS) being developed by the FBI (Federal Bureau of Investigation). Virtually every new system that must process large volumes of data or complete the processing in a small amount of time is likely to rely on some form of parallel processing. The problem with multicomputers is that they are inherently much harder to program than single-computer systems. Programming a multicomputer application may require managing the synchronization among tens, hundreds, or even thousands of independent processes that must be coordinated to solve a single problem. This project seeks to make programming such computers significantly less difficult, thus reducing the cost and risk of using them in government systems.

A major concern is the programmer's ability to manage the synchronization required by a large complex program. Perhaps the interactions among the program elements is very complex, perhaps it makes use of code written by others, or perhaps synchronization at particular points in the program depends on input data. Debugging multicomputer programs is generally more difficult than single-computer programs, because they may exhibit intermittent errors due to slight timing differences when two or more threads of control access the same memory location in an unsynchronized manner. When enhancements are made to a multicomputer program, errors may arise caused by timing differences introduced by the enhancements. In an attempt to address these issues, we are developing a multicomputer run-time executive called the Synchronization Manager (SaM). SaM makes it easier to develop, debug, and enhance multicomputer software, because it automatically manages the synchronization required by an object-oriented program to produce the same results as a single-computer execution. Timing differences have no effect on the results produced by an application program executed on a multicomputer using SaM. In addition, SaM can exploit input data-dependent concurrency that can only be identified at run time.

Object-oriented programming is a good model of computation for distributed-memory, message-passing multicomputers, because it minimizes global information and provides natural communication and synchronization boundaries. When writing programs for a multicomputer, it is a good idea to assign data items and code that will be used together to the same processor. When writing an object-oriented program, the programmer divides the data and the functions to control that data into objects. A side effect of this assignment is that data and functions that will be used together are identified. Thus, object-oriented programming facilitates the automatic mapping of software to hardware performed by SaM. Most importantly, these are a natural part of the object-oriented programming model of computation.

The synchronization manager uses future objects and checkpoint and rollback to generate and manage synchronization. Context objects manage method execution, including handling recursive cycles of messages correctly. There are a number of objects which comprise the SaM run-time executive. These objects manage synchronization and communication for the application program. SaM cannot run ordinary C++ application programs; memory management and error handling must be carefully controlled in SaM. However, our application programming language, CPM, is syntactically similar to C++. In [Prelle:92], we described the synchronization manager and the application language in detail. We also described our results when we tested SaM using a synthetic application program generator to run application programs written in CPM on a Symult S2010 multicomputer. In addition to testing the correctness of the SaM implementation, we have also shown the basic correctness of the synchronization manager model mathematically. [Bridgland:91] contains the details of the mathematical proofs, [Prelle:90] contains a more intuitive, descriptive exposition of the arguments and the results. In [Brando:92], we described a fault-tolerant Lisp language implementation of SaM.

In this paper, we will briefly describe the synchronization manager as well as the enhancements made to SaM and CPM this year to improve performance and provide language constructs to allow the application program developer to control synchronization directly. We will also describe our experience using SaM to run several application programs—image perspective transformation, neural network training, and multiple target tracking—written in CPM, on a Symult S2010 multicomputer.

SECTION 2

THE BASIC SYNCHRONIZATION MANAGER

The run-time behavior of a program with a given input data set can be represented as a directed graph with cycles (figure 1). The circles in this figure represent objects, and the arrows represent messages. When an object receives a message, the method associated with that kind of message begins executing. The labels on the arrows show the order in which messages are processed when we execute this program on a single computer. Every arrow represents a request for processing. Associated with each request is a reply message that is not shown.

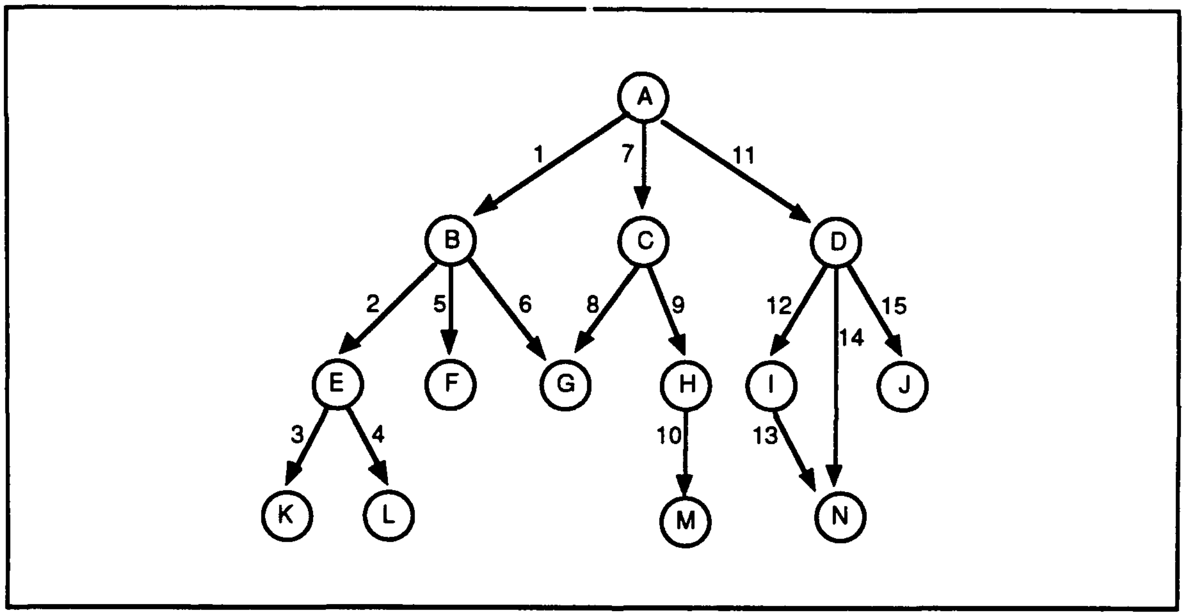

Figure 1. Serial Execution

In SaM, each object in the system that requires synchronization services is associated with an instance of the synchronization manager class. We refer to all instances of this class as *the* synchronization manager.

SaM uses data-driven synchronization and speculative computation to gain concurrency, and checkpoint and rollback to ensure the same results as a single-computer execution. Data-driven synchronization means a process does not block until it needs the result of another process' computation. SaM uses future objects to support data-driven synchronization in basically the same way they are used in the Actor model of concurrent object-oriented programming [Agha:86]. In Actors, ES-Kit [Chatterjee:89], and MultiLisp [Halstead:85], it is the programmer's responsibility to manage futures; in SaM, they are created and managed automatically.

On a single-computer system, every object processes one message at a time in a fixed order (dependent on the input data). On a multicomputer, messages may arrive at an object in a different order, even with identical input data. Speculative computation means that an object processes available messages in the correct order, but without regard for messages not yet delivered, even if those late arriving messages would have been processed earlier in a sequential execution.

To ensure the same results as a single-computer execution, we generalized the Time Warp synchronization mechanism that was developed for distributed object-oriented discrete-event simulation [Jefferson:87] and [Reiher:90]. In both Time Warp and SaM, an object's state is saved, or checkpointed, whenever it processes a message. If an object has processed a message out of order, it is rolled back to the appropriate state, and messages are reprocessed in the correct order. It may be that as a result of processing messages out of sequence, erroneous messages were sent to other objects. During rollback *negative* messages are sent to retract erroneous *positive* messages. When an object receives a negative message, there are three possibilities. If the matching positive message is among the object's unprocessed messages, the two messages cancel each other. If the positive message has not been received yet (this is possible in a network that employs adaptive routing [Chow:87]), the object's synchronization manager saves the negative message so that it can cancel with the positive message when it arrives. If the positive message has already been processed, the object's synchronization manager rolls the object back to the state prior to the one the positive message was processed in. The negative message then cancels with the positive message. A negative message is an *antimessage* for its positive message, and a positive message is an antimessage for its negative message.

In Time Warp, it is the responsibility of the application program to generate timestamps that are used to order messages properly; SaM generates these timestamps automatically. An advantage of both Time Warp and SaM is that deadlocks and races cannot occur. This reduces the difficulty of developing software for multicomputers considerably.

COMPUTATION TIME

In discrete-event simulation, simulation time is used to determine the order that events (messages) should be processed in, and the application program is responsible for associating a timestamp with each message an object sends. In general-purpose computation there is no sense of time, but there is a sense of order. So we developed a mechanism that works like simulation time to indicate the order that computation events should be processed in. The synchronization manager associates a string of characters with each message an object sends. This character string indicates the order that the message would have been processed in if the computation had been executed sequentially. Since the role of these character strings is similar to timestamps used in discrete-event simulation, we call them timestamps also.

When an object processes a message with a given timestamp, its synchronization manager appends a character to that timestamp for every message the object sends. In figure 2, for example, the object B receives a message with timestamp "a". The timestamp on the first message that B sends while processing this message is "aa", the timestamp on the second message is "ab", then "ac", and so on. Notice G's synchronization manager can recognize that the message from B should be processed before the message from C because "ac" is less than "ba" lexicographically. However, G speculatively processes whichever message arrives first. Before any message is processed, the synchronization manager saves G's current state. Thus, if the "ba" message arrives and is processed before the "ac" message, G can be rolled back to its previous state when the "ac" message arrives so that "ac" and "ba" can be processed in the correct order.

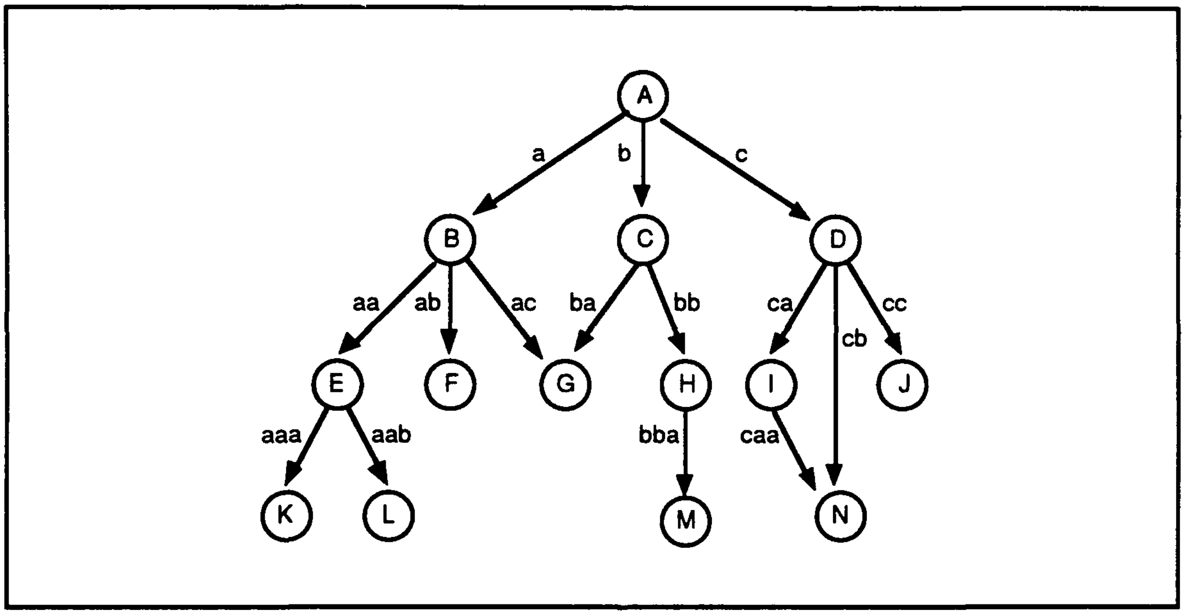

Figure 2. Event Order Synchronization

In general, an object processes one message at a time. Recursive cycles of messages are an exception to this rule. In a recursive cycle, an object must process one message in the midst of processing another message, for example, a recursive cycle can be established directly when an object sends a print message to itself in the middle of executing a method. It is also possible for a recursive cycle to be established indirectly through another object. When a recursive cycle of messages is recognized, an object processes more than one message at a time, but the processing is serialized in the same way as in a conventional sequential execution.

The synchronization manager can recognize when an object has received a message that is part of a recursive cycle of messages, because the timestamp on the object's current state will be a prefix of the timestamp on the recursive message (following any rollback that may be necessary as a result of receiving the recursive message). In figure 3, we see that there are four objects that send messages to M: H, I, J, and N. Suppose that the processing of the message from I to M results in M sending a message to N, and N's processing of that message causes N to send a message to M, that is, a recursive cycle is established. However, let us also assume that none of the other messages to M causes a recursive cycle to occur.

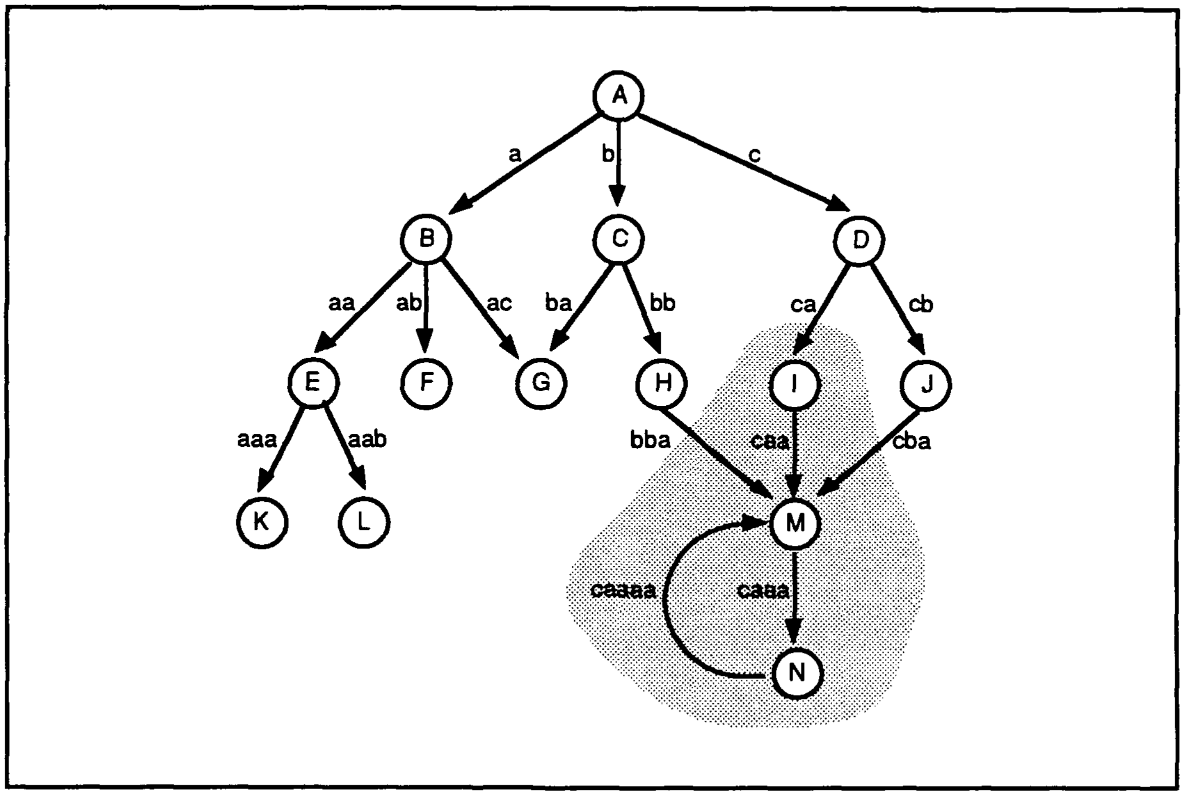

Figure 3. Recursive Message Cycle

M's synchronization manager has to be able to identify three different situations: a message with an earlier timestamp than the one it just processed or is currently processing; a message that indicates a cycle is about to take place; a message with a later timestamp that is not part of a cycle. If the timestamp of the message that M is processing is greater than the timestamp of the incoming message, then M must be rolled back. In the example, if M is processing the "caa" message from I when it receives the "bba" message from H, M must be rolled back.

If the timestamp of the message that M is processing is less than the timestamp of the incoming message, then M's synchronization manager must decide if the incoming message should be processed after the current message has been completely processed or if a recursive cycle is taking place. In the example, if M is processing the "caa" message from I when it receives the "cba" message from J, M's synchronization manager should keep this message queued until M has completely processed the "caa" message. On the other hand, if M receives the "caaaa" message from N, it must recognize this as a recursive cycle which must be processed before the completion of the "caa" message. M's synchronization manager can recognize that a recursive cycle is occurring because the timestamp of the current message "caa" is a prefix of the timestamp of the incoming message.

The width of a timestamp depends on two factors: the maximum number of messages an object can send in the course of executing a single method, and the maximum depth of a method invocation sequence (for example, A sends a message to B, which in turn sends a message to C, etc.). The latter is equivalent to the maximum number of stack frames on a processor's control stack at any given time in a single-computer execution. The 256 ASCII character set permits an object to send as many as 256 messages during the execution of a single method. For some applications, this may be insufficient. Thus, we have implemented timestamps as an array of short integers (16-bit) also. In this implementation, timestamp comparison is performed by comparing individual elements of the arrays.

Thus, given timestamps A and B (|A| = the length of timestamp A, and |B| = the length of timestamp B), their relationship is defined as follows:

$$
\begin{aligned}
A = B &\Leftrightarrow |A| = |B| \wedge \forall i \in [0,|A|)(A_i = B_i) \\
A < B &\Leftrightarrow |A| < |B| \wedge \forall i \in [0,|A|)(A_i = B_i) \\
&\qquad \vee \exists j \in [0, \min\{|A|,|B|\}) \text{ such that } \forall i \in [0,j)(A_i = B_i) \wedge A_j < B_j
\end{aligned}
$$

This implementation is similar to the Dewey decimal timestamps employed in ParaTran [Tinker:88].

GLOBAL VIRTUAL TIME

A disadvantage of a rollback scheme is that a good deal of memory can be used to save old states and messages. Time Warp uses the concept of *global virtual time* (GVT) to reduce the amount of information that must be kept. The essential idea is that the computation is always moving forward. Thus, there is a simulation time (or, in our case, a computation time) past which the computation can never roll back. This time is the GVT, and ways exist for computing a safe approximation to GVT dynamically while computation is proceeding [Jefferson:87, Samadi:85, Bellenot:90].

We can think of GVT as the commit time of the computation. After a GVT estimate is computed, states and messages with earlier timestamps can be discarded. Speculative computation can cause the application to commit errors (e.g., divide by zero) that a sequential execution would not have committed. Eventually, rollback will undo such errors. Thus, application output and errors can only be committed when GVT has passed the point in the computation when they were generated.

SECTION 3

APPLICATION LANGUAGE

SaM cannot run ordinary C++ application programs, because memory management and error handling must be carefully controlled in SaM. Our language, CPM (C Plus Minus—C plus objects, minus pointers), is syntactically similar to C++. Programs written in CPM can be executed either sequentially without the synchronization manager or in parallel using SaM. If the program is to be executed sequentially, a translator is used to translate object-oriented CPM code to ordinary C code. If the program is to be executed using SaM, a different translator is used to translate the user's code into a form that will allow SaM to manage its execution.

CPM supports two different types of classes: concurrent classes and local classes. Concurrent objects may be distributed physically in the system. Their synchronization is managed by SaM, that is, each of them has a synchronization manager object associated with it. Concurrent objects have object addresses associated with them. When a concurrent object is passed as an argument, its object address is passed. Circular referencing among concurrent objects is supported. When we spoke of application objects previously, we were describing concurrent application objects.

A local object may be the value of an instance variable of a concurrent object or another local object. A local object is always part of the state of one and only one concurrent object. Thus, their synchronization is managed by their concurrent object's synchronization manager. When a local object is passed as an argument, it is passed by value. The translator automatically generates copy functions to take care of packaging up local object arguments. Pointers to local objects and circular referencing among local objects is not supported (if circular referencing is required, concurrent objects must be used instead).

All basic C data types are supported such as int, char, long, float, double, and statically allocated arrays. In order to make building application programs easier, we are implementing a library of commonly used local classes. Another reason for building these local classes is that part of their implementation is outside the scope of the application language as defined and needs special hand coding. Among the local classes to be implemented are strings, linked lists (parameterized by type), and doubly linked lists (also parameterized by type). These classes will also be robust, checking for errors and boundary conditions. Global constants are allowed in the system, but not global variables. Global variables are not safe since their synchronization is not managed by SaM.

Because SaM allows objects to process messages out-of-order with respect to a sequential execution, errors may occur in a SaM execution which would not have occurred in a sequential execution. If the error is caused by processing messages out-of-order, it must be possible to recover from the error by simply rolling back when the appropriate message or messages finally arrive and are processed in the correct order. On the other hand, if the error is a real application error (that is, it would have occurred in a sequential execution), then it can be committed to when GVT reaches the timestamp on the state when the error occurred.

For example, errors that can be trapped by the system (divide-by-zero) send a signal to the executing process. When an error signal is caught, the context is put in an error state. When a context is in an error state, its synchronization manager will accept messages but will not allow the context to execute. Eventually either a message will be received that causes the context to be rolled back to a non-error state, or GVT will reach the timestamp of the error state and the computation will be aborted.

Because an error may be an artifact of speculative computation and not the application program, we cannot allow application behavior to endanger the run-time system or other parts of the application program. Since the application and SaM share the same memory, if we allowed pointers to local memory in our language as C and C++ do, it would be possible to write into memory that is not part of the object's currently executing environment. If this type of error occurred, it would not be corrected by rollback. Thus, each object's state must be managed separately. For arrays, index out-of-bounds errors are captured before they occur.

Interactions among concurrent objects are controlled by the SaM runtime system. The interface points between the application method execution and the run-time system include: creating an instance of a new concurrent object, sending an application message to a concurrent object, printing a result, resolving a future, and completing execution which may include sending a return value to a future.

To support recursive cycles of messages, error handling, and future processing, it must be possible to suspend and resume method execution (possibly to a previous state) when appropriate. We would like to treat method execution as a light weight process. Since LWP libraries are not available on the iPSC/2 or the S2010 (and there are bugs in the Sun LWP code), we implemented a mechanism that is similar to light-weight processes using *setjmp* and *longjmp* system calls. To save the state of an executing method, we must save all registers, local variables (the stack), instance variables of the object, and method arguments. *setjmp* suffices to save all registers, but does not take care of saving anything on the stack. We added code to save the appropriate amount of stack as well as instance variables and method arguments to resume execution. The saved state of a method execution is called an environment.

The interface between the application language and the runtime system is controlled by code inserted by the preprocessor into the application code. This interface code calls the appropriate synchronization manager services. The context's synchronization manager performs the appropriate service and then determines whether or not to resume method execution immediately. If method execution cannot be resumed immediately, a copy of the environment is saved. Since an application message sent by a context can lead to a recursive cycle of messages being established, a copy of the environment is saved when a message is sent to another concurrent object, even if method execution is resumed immediately.

SECTION 4

SYNCHRONIZATION MANAGER ENHANCEMENTS

FREEING OBJECTS DYNAMICALLY

Context objects may be freed when they are no longer needed by the application. Such contexts can be recognized when they have finished executing or have been rolled back to their initial state and the timestamp on their last state is earlier than GVT.

The application may call for other application objects and multifutures to be freed when they are no longer needed by the computation just as data structures are freed in C. An object can be safely freed when GVT is greater than the timestamp on the free message.

BALANCED TREES

We implemented balanced trees for managing the harness' association list of global object addresses and objects. Because keeping the tree balanced is an expensive process, it does not really speed up SaM more then a simple list implementation unless there are a very large number of objects being managed by a harness.

MULTIFUTURES

We have implemented the sam_multifuture class. In our original model, a future was designed to hold one value—the result of a single computation. There are times when the user may want to start a number of computations, but can use any one of the results as soon as it is available. We can accomplish this by creating a special kind of future, a *multifuture*, that is able to hold a number of values, rather than just one. Our multifutures are similar to future-sets in ES-KIT [Chatterjee:89]. Multifutures may return the results they hold one at a time or in groups as soon as they are available. Unlike futures, multifutures are created and values are retrieved from them under programmer control. They provide an explicit parallel extension of our application language, CPM.

We implemented two kinds of multifutures in the Lisp version of SaM: ordered and unordered multifutures [Prelle:91]. Ordered multifutures return the results of the computations they hold in timestamp order, thus they ensure that the computation will produce the same results as a sequential execution of the same program. They serve to reduce the number of futures that must be created and the amount of message traffic that results when many futures must be resolved individually. Unordered multifutures return the results of the computations they hold in the order they are received, which may not correspond to timestamp order. Thus, there is no guarantee that the results of a parallel execution will correspond to a sequential execution. In the C-based implementation of SaM, we implemented ordered multifutures only.

An ordinary future may receive only one *relevant* (ultimately not rolled back) set_value message. Multifutures usually receive many set_value messages. In addition, multifutures may receive get_values and set_holds messages. The set_holds message indicates how many values the multifuture will ultimately hold.

As shown in figure 4, object R creates multifuture MF. R sends messages to other objects requesting that they send the results of their computations to MF. R's synchronization manager keeps track of how many request messages were sent. When R has finished sending messages using MF, R's synchronization manager sends a set_holds message to MF indicating the number of values MF will ultimately hold.

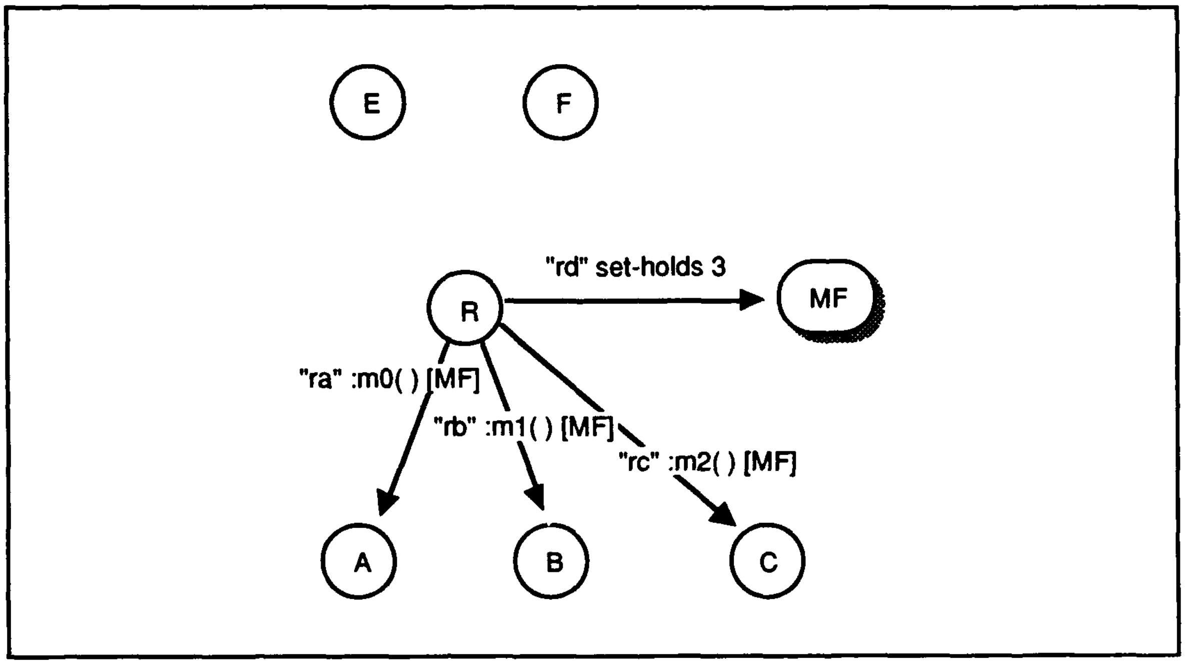

Figure 4. Multifuture Creation

R may pass the global object address of MF to other objects in the system, for example, the objects E and F in figure 5.

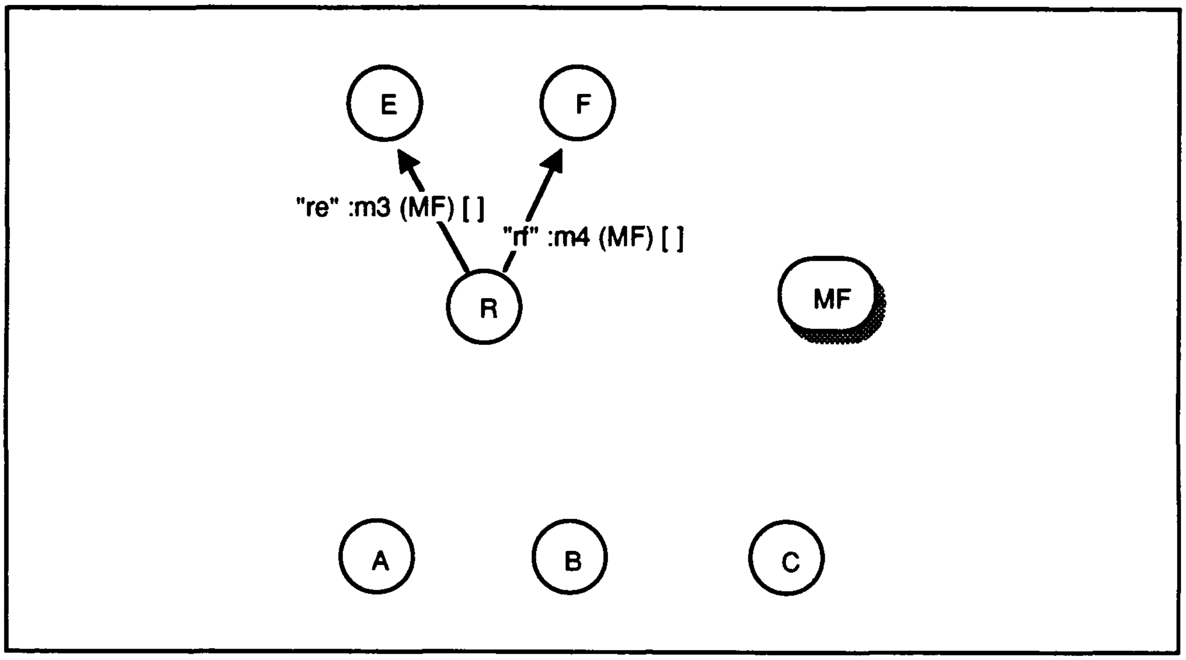

Figure 5. Multifutures as Arguments

When A, B, and C have completed their computations, they send their results to MF in set_value messages, just as they would to any future (figure 6). Objects that know MF's global object address, like E and F, may retrieve values from MF in two different ways. If a get_values message to a multifuture has two numerical arguments, then all values between those specified are required by the requester. For example, because the get_values message from E has the arguments 0 and 1, E is asking MF to send its first (the one from A) and second (the one from B) values. MF will not reply to E until both its first and second values are available. If a get_values message to a multifuture has one numerical argument, then the requester is asking the multifuture to send it all the available values starting from the one specified. For example, F is asking MF to return all values available starting with the first. MF will return all values available starting with the first value requested. If the first value F requested is not available, MF will hold F's request until it is.

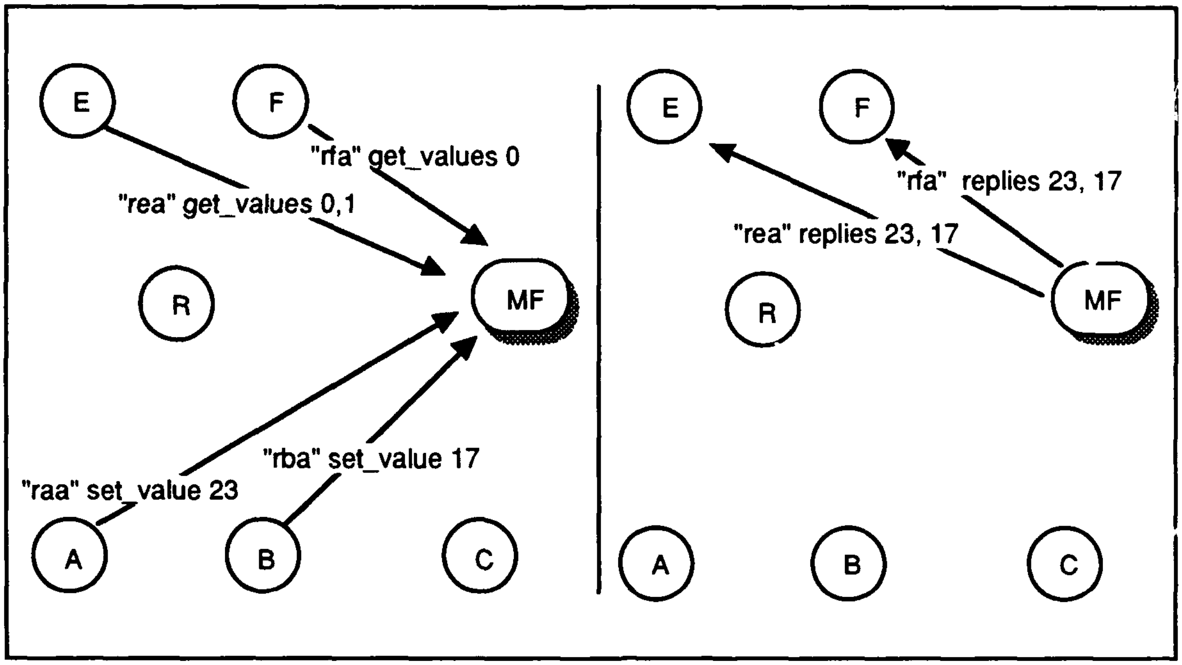

Figure 6. Setting Multifutures and Getting Their Values

Multifutures process get_value messages in other than strict timestamp order. We decided to distinguish two situations: before GVT passes the timestamp on the set_holds message and after GVT passes the timestamp on the set_holds message. Before GVT passes the timestamp on the set_holds message, set_value messages may be withdrawn. This means that get_values messages may have to be rolled back and reprocessed. Table 1 shows the old_states stack of a multifuture before GVT passes the timestamp on the set_holds message. Note, the old_states is a stack, therefore, states that appear lower in the stack were created before states that appear higher in the stack.

Table 1. Old_states Stack of a Multifuture

Message Received	Timestamp	Values	Holds	Message Sent
"rd" set_holds 3	"rd"	23 17 58	3	
"reb" get_values 2 2	"reb"	23 17 58	—	"reb" Replies 58
"rfb" get_values 2	"rfb"	23 17 58	—	"rfb" Replies 58
"rca" set_value 58	"rca"	23 17 58	—	
"rfa" get_values 0	"rfa"	23 17	—	"rfa" Replies 23 17
"rea" get_values 0 1	"rea"	23 17	—	"rea" Replies 23 17
"rba" set_value 17	"rba"	23 17	—	
"raa" set_value 23	"raa"	23	—	
	"ra"	—	—	

After GVT passes the timestamp on the set_holds message, all set_value messages have been committed to, so the MF need never roll back again. get_values messages may be processed in any order. If a negative get_values message arrives, the multifuture does not roll back, it simply withdraws the associated reply.

Table 2. Old_states Stack of a Multifuture

Message Received	Timestamp	Values	Holds	Message Sent
"xmb" get_values 2 2	"xmb"	23 17 58	3	"xmb" Replies 58
"vfa" get_values 0	"vfa"	23 17 58	3	"vfa" Repiies 23 17 58
"xma" get_values 0 1	"xma"	23 17 58	3	"xma" Replies 23 17
"wqb" get_values 0	"wqb"	23 17 58	3	"wqb" Replies 23 17 58
"rd" set_holds 3	"rd"	23 17 58	3	

When a context sends a get_values message, it blocks method execution until it receives a reply. Thus, such an object never has more than one outstanding get_values at a given time. Therefore, the timestamp on each get_values sent by the same context will arrive and be processed in timestamp order with respect to each other.

ASSERT METHOD

An assert method is a technique that the programmer can use to cause synchronization barriers. The role of the assert method is much like that of a guard in Argus [Liskov:88] or ABCL/1 [Yonezawa:87] based on communicating sequential process model described in [Hoare:78]. The programmer can place synchronization barriers in the code to prevent the synchronization manager from spawning unnecessary parallel tasks that will be rolled back. For any concurrent class message, the programmer can define an assert method which returns a Boolean value indicating whether or not the message should be processed, given the current values of the instance variables and arguments. The assert method may look at instance variables and arguments to the method, but it may not cause any futures to be resolved or send any messages to concurrent objects.

The object's synchronization manager executes the assert method before processing the associated method. If the assert method returns TRUE, the message is processed. If the assert method returns FALSE, the message is not processed but remains in the input queue.

GRAPHICAL FRONT-END

The graphical front end (GFE) supports application program development. The GFE displays information about program execution as it proceeds, and can only be run on the multicomputer simulator. There is an object status display window that shows an icon for every time-managed object. Five kinds of objects are distinguished by differently shaped icons: system objects (creator objects and the output object), application objects, contexts, futures, and multifutures. In addition, futures and multifutures change shape when they have been set. Each icon is displayed in a color that indicates the current state of the corresponding object. These states include idle, enqueuing message, rolling back, processing message, busy, ready to execute, executing, future waiting, multifuture waiting, recursion waiting, error, and complete. A separate icon key window identifies the significance of each icon shape and color.

We also implemented a processor utilization window, which shows, for each iteration of the main simulation time-step loop, the fraction of maximum possible processing (message delivery, object creation, and context execution) accomplished on each simulated multicomputer node. The object status display and processor utilization windows are useful for visualizing how an application's execution is proceeding and, in particular, the degree of parallelism achieved at each step during the execution.

The user may click on an icon in the object status display window and receive information about the icon and its corresponding object (for example, the object's name and global address) in a message window.

SEND TO SELF OPTIMIZATION

The compiler was enhanced to optimize method invocations to "self". Previously, a new context was established for every method call whether or not it was a send to self. However, since a send to self cannot cause any rollbacks to occur, it is sensible to make send to self a special case. Therefore, instead of treating a send to self like a concurrent class method invocation (a send request), the compiler treats this special case like a local class method invocation (it simply calls the method instead of having the synchronization manager intervene which would create a context and extra states). This optimization completely eliminates the overhead of send to self that was previously incurred by synchronization manager intervention.

GVT

We implemented a new version of the GVT calculation that avoids sending acknowledgments based on [Bellenot:90]. The new GVT requires a three phase protocol. First every object is told to prepare for a GVT calculation to begin. From this point on each object keeps track of the minimal timestamp on the messages it sends. Then each object is told to calculate OVT (Object Virtual Time). For an application object, a creator, a future, or a multifuture, OVT is the minimum of the timestamps on the messages in the object's input queue and the minimum timestamp of the messages it sent since it was told to prepare for a GVT calculation. For context objects, the timestamp of the current state is considered also if the object has not completed execution. Finally, each object is assigned the new GVT. This protocol works properly as long as a sufficient amount of clock time is allowed to pass between the prepare phase and the calculate GVT phase so that all messages sent prior to the prepare phase are delivered.

In our implementation, there is a gvt_master on node zero and gvt_controllers on all nodes participating in the computation. The gvt_master is invoked based on the last time

GVT was calculated. If the last GVT computation is complete, a new GVT calculation begins when the gvt_master sends a prepare_gvt message to each gvt_controller. The gvt_controllers asynchronously notify the objects on their respective processors that they should prepare for a GVT calculation to begin. Based on the time GVT prepare phase began, the gvt_master is invoked to begin the actually GVT calculation phase. The gvt_master sends calculate_gvt messages to each gvt_controller. The gvt_controllers asynchronously poll the objects on their respective processors and send the minimum timestamp calculated to the gvt_master. When all controllers have reported, the master sends the minimum timestamp received in an assign_gvt message to each controller. The controllers then assign the new GVT to each of the synchronization managed objects on their processors. Each object drops states with timestamps less than the newly assigned GVT.

SECTION 5

IMAGE PROCESSING APPLICATION

The image processing application we are using contains code to do three-dimensional perspective transformation including hidden-line elimination. It is based on code found in [Ammeral:86]. The application reads in a collection of vertices and polygons that describe a three-dimensional image. Each vertex is a set of three real numbers that represents the x, y, and z coordinates of a point in three-dimensional space. Each vertex has an integer associated with it that is used to identify it uniquely. Each polygon is a list of two or more integers (associated with vertices) representing either a simple line or the faces of a three-dimensional solid in the image.

The application then reads in one or more sets of angles that indicate different views of the image (view angles). Hidden-line elimination is performed by using the view angle set to construct triangles from the polygons. Each line in the image (given by the polygon connections) is compared to each triangle to determine if the triangle hides the line completely, partially, or not at all. If the line is only partially hidden by a particular triangle, the parts of the line that are not hidden are tested against other triangles recursively. The output of the program is a set of commands that another program can use to draw the image in two dimensions from the perspective given by the view angle set. Each command consists of two floating point numbers and a command: an "m" for move, a "d" for draw, or an "e" for erase the screen.

The output of the program for one view was used by another program to draw the image shown in figure 7. The output of the program for more than one view can be used by the other program to produce a "movie" that shows the image viewed from different angles. Ten capital letter A's in the image gives an image with 200 vertices; eight gives an image with 160 vertices; twelve gives an image with 240 vertices.

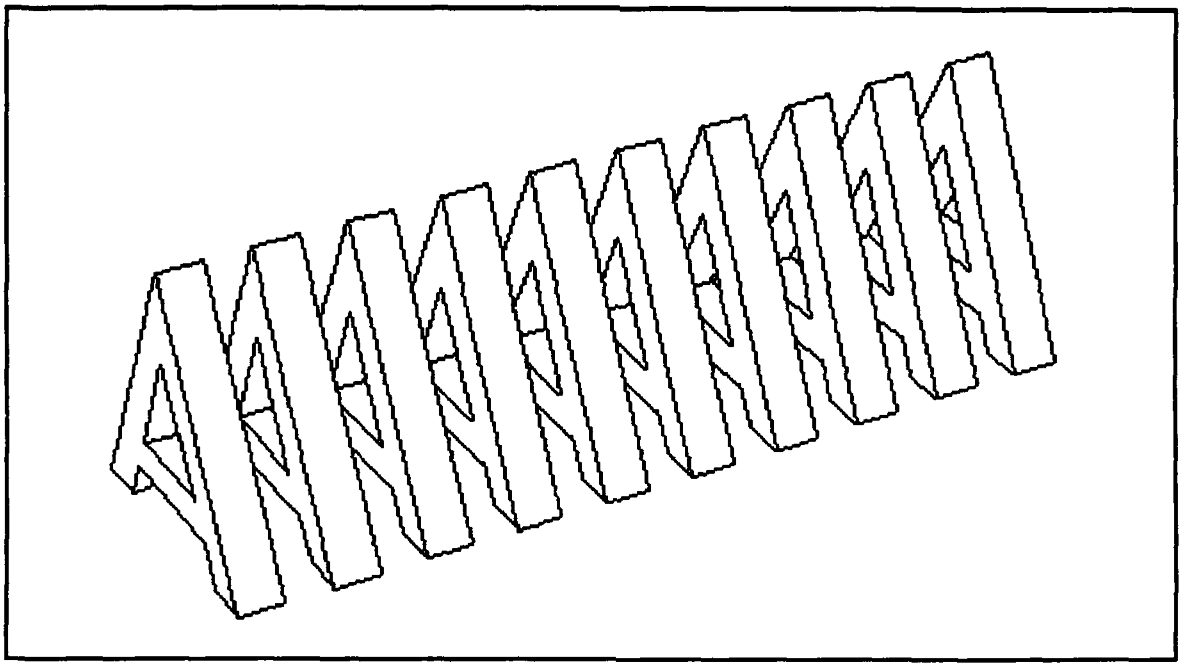

Figure 7. Image with 200 Vertices

The code presented in Ammeral's book was not object-oriented. So our first task was to translate it into an object-oriented form. In our first implementation, each vertex and polygon was a concurrent object. The granularities of the computations (number of instructions executed per communication event) were too fine compared to the overhead incurred by communication using SaM. We modified the application to contain more coarse-grained parallelism than was present in the first implementation. We did this by creating big concurrent objects each of which has a copy of all the vertices and polygons as local objects. The start object reads in all the vertices and polygons. Using this information, it creates the big objects, one on each available processor (figure 8).

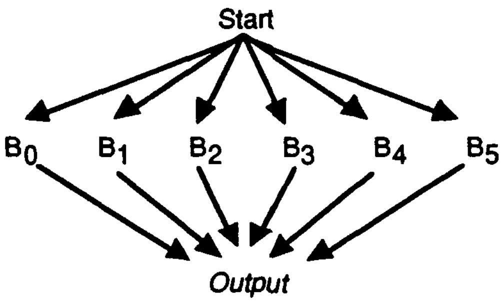

Figure 8. Perspective Transformation Control Flow

We tried two different implementations based on this approach. In one implementation, each big object computes a different view of the image, that is, the start object sends each big object a different view angle set. Using the view angle set, each big object computes the triangles for the set of polygons and performs the hidden line computation for its own view angle set in parallel. In this implementation, when the number of view angles matches the number of nodes used for big objects, performance is roughly the same as computing one view angle set when printing is suppressed (figure 9).

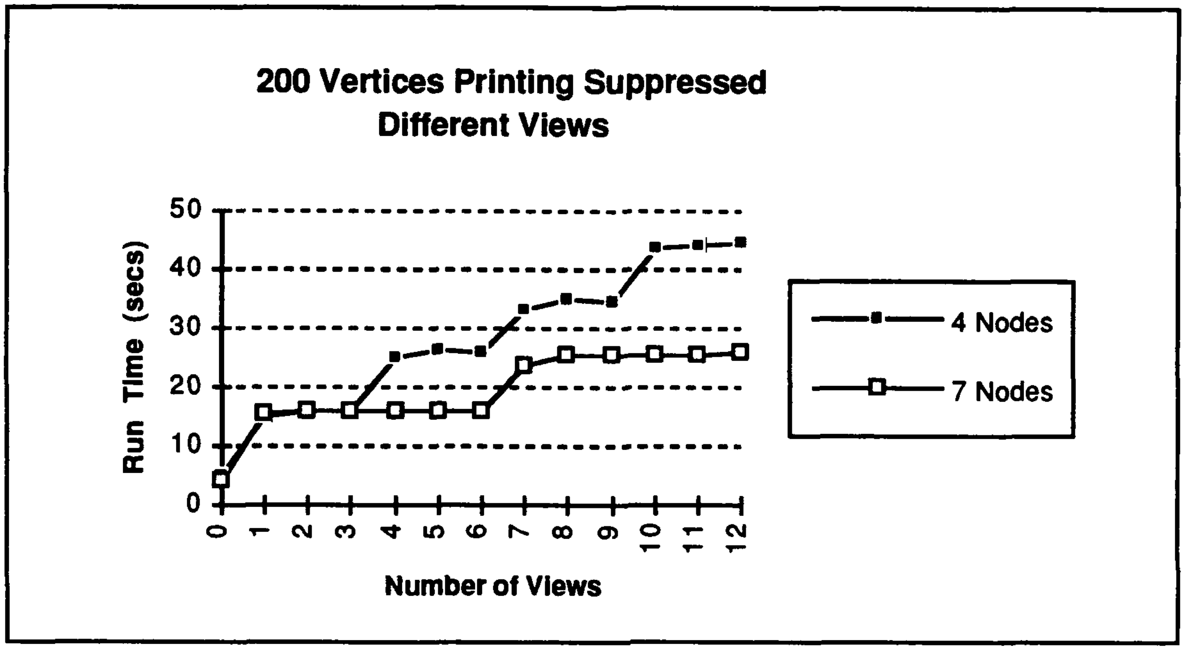

Figure 9. 200 Vertices Printing Suppressed Four and Seven Nodes Different Views

We note that an interesting phenomenon occurs in this implementation when printing is enabled (figure 10). The figure suggests that when printing is enabled on seven nodes, the run-time increases non-linearly with the number of views. On the other hand, in a four node implementation, when printing is enabled the run-time appears to increase linearly with the number of views. We will investigate this phenomenon in greater detail in the implementation we will describe next.

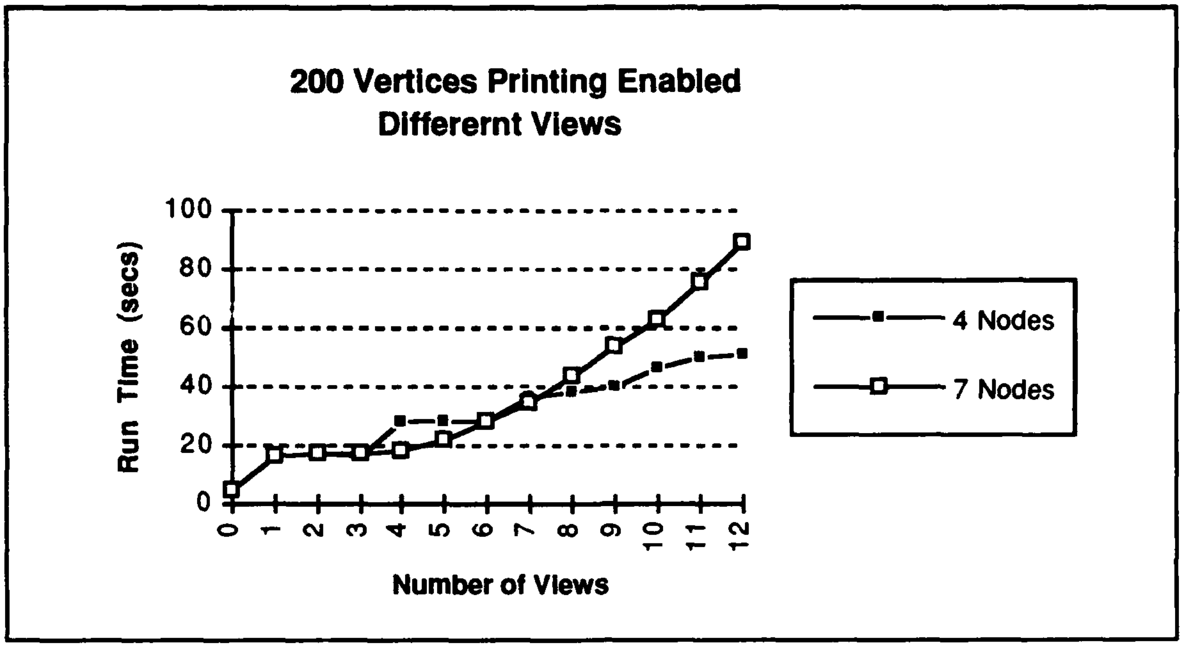

Figure 10. 200 Vertices Printing Enabled Four and Seven Nodes Different Views

Although this implementation performs reasonably well when the number of views is a multiple of the number of nodes used for big objects, some processors are idle when this is not the case. An approach that distributes the computation more evenly among the available processors seems more desirable.

In the approach we will consider for the remainder of this section, the start object sends every view angle set to each big object. Using the view angle set, each big object computes the triangles for the set of polygons. As part of the triangle computation, potential lines to be drawn are identified and each line is associated with one of its vertices. Each vertex is associated with an integer. Each big object checks the lines that are associated with a subset of the vertices to determine if it is hidden or should be drawn. So that each big object gets approximately the same amount of work to perform, the lines to be checked by a big object B_i are those associated with the vertices v_j where j modulo the number of big objects is equal to i. Each big object functions independently from the others. However, the output from different view angle sets must be kept in the proper order and must not be mixed together so that they can be read by another program to form a "movie" of the image being rotated.

We will examine how SaM executions perform against sequential executions of this application. There are three variables of interest: number of vertices, number of views, and number of nodes. We will examine the functional relationships among these variables.

Figure 11 shows the run-time˒ for the sequential version of the code executed on one S2010 node without SaM and two parallel versions using SaM on four S2010 nodes. All the versions used three big objects. The sequential execution was performed with output suppressed because when output was enabled an error occurred while printing floating point numbers. We believe the cause of the difficulty was the small amount of memory devoted to stack space (which has a fixed limit of 32K bytes) on each S2010 node. The only sequential case where we were able to obtain output was for an image with only 40 vertices. Each of the SaM versions shown used four nodes with three big objects (3) each placed on a different S2010 node. Printing floating point numbers was not a problem in the SaM version. So we show the results of executing two versions: one with printing suppressed (np) the other with printing enabled. As can be seen from the figure, as the number of views increases the performance of both SaM versions is better than the sequential version.

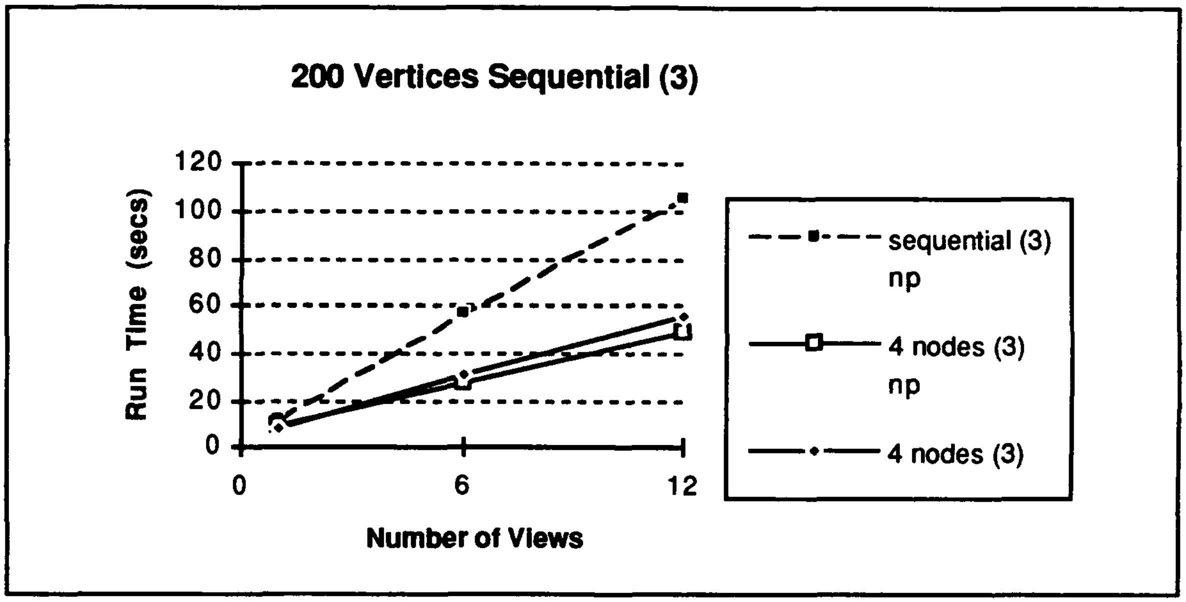

Figure 11. Perspective Transformation 200 Vertices Sequential (3)

In the parallel execution, each big object performs the computation necessary to divide the polygons into triangles separately to avoid communicating this information. In the sequential execution, it is not necessary to have this computation performed more than once. To avoid computing the triangles more than once, one big object can be used rather than three. Figure 12 shows the run-times when only one big object is used for the sequential execution. As can be seen from the figure, the run-time of both SaM executions is increasing linearly as the number of views increases but at a slower rate than the sequential execution (0.5 times the sequential rate of increase when printing is enabled in SaM but not in the sequential version, 0.4 times the sequential rate of increase when printing is suppressed in both).

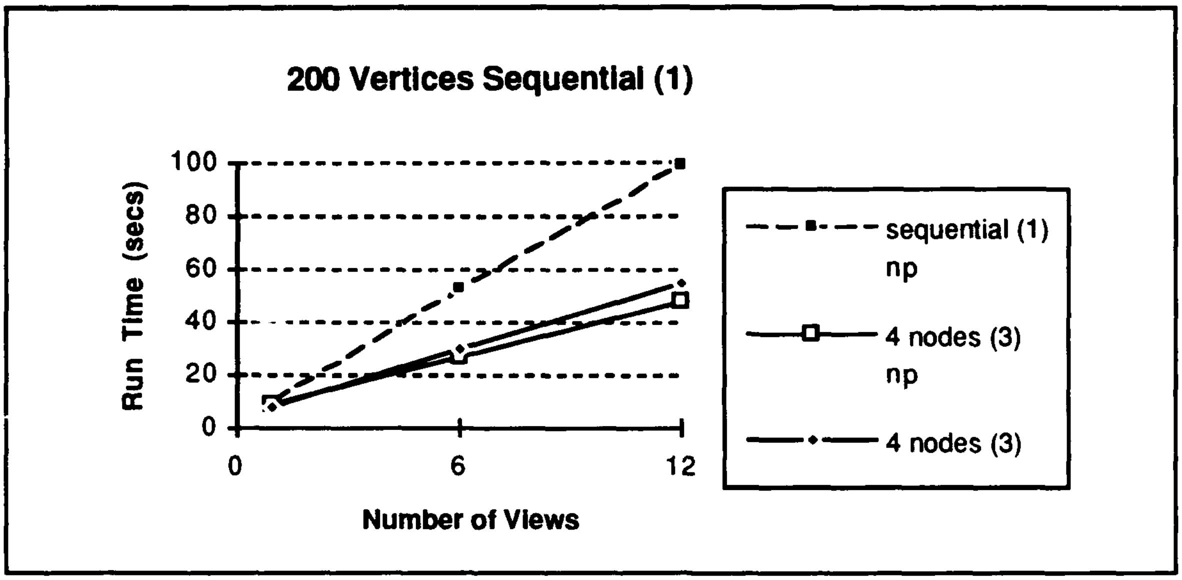

Figure 12. Perspective Transformation 200 Vertices Sequential (1)

Figures 13 and 14 show similar graphs but with the number of vertices varied. As can be seen from the figures, the run-time of both SaM executions is increasing linearly as the number of views increases but at a slower rate than the corresponding sequential executions.

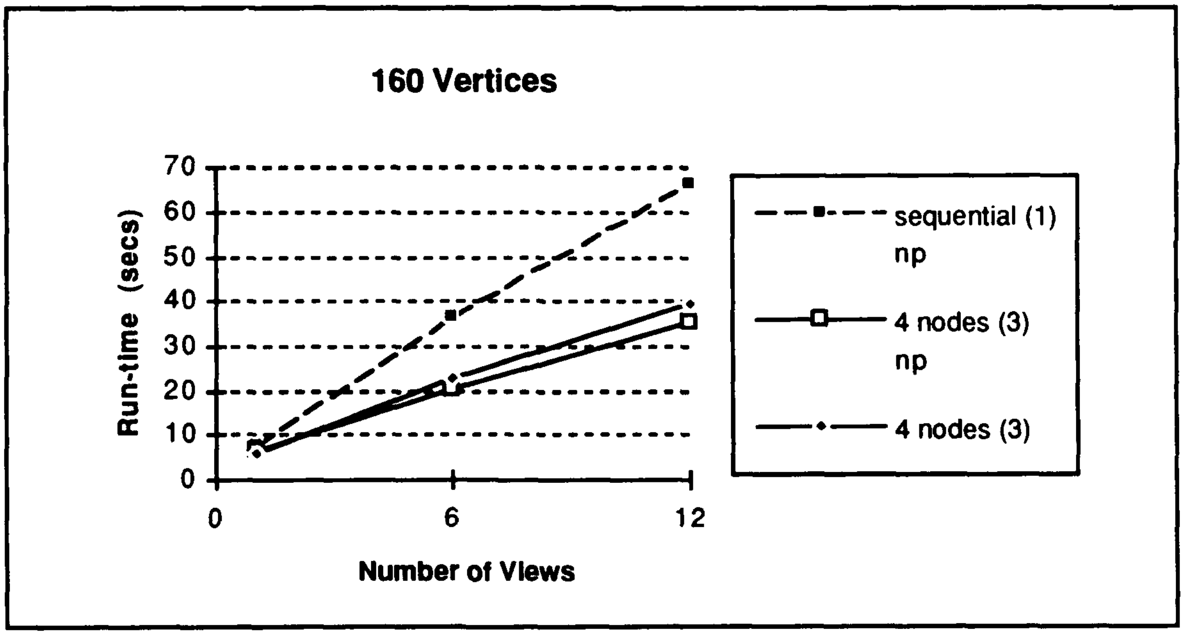

Figure 13. Perspective Transformation 160 Vertices

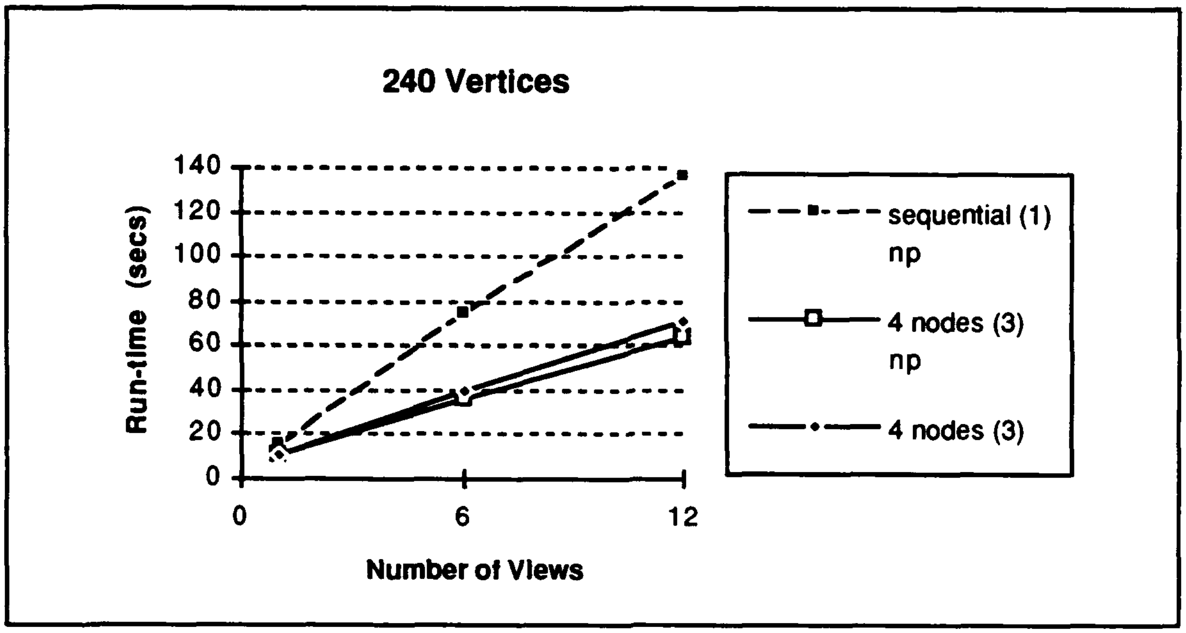

Figure 14. Perspective Transformation 240 Vertices

Figure 15 shows a comparison of the run-times when the number of vertices and the number of views are varied. Since the image is more complex as the number of vertices increases, the computation is more complex and the run-times increase.

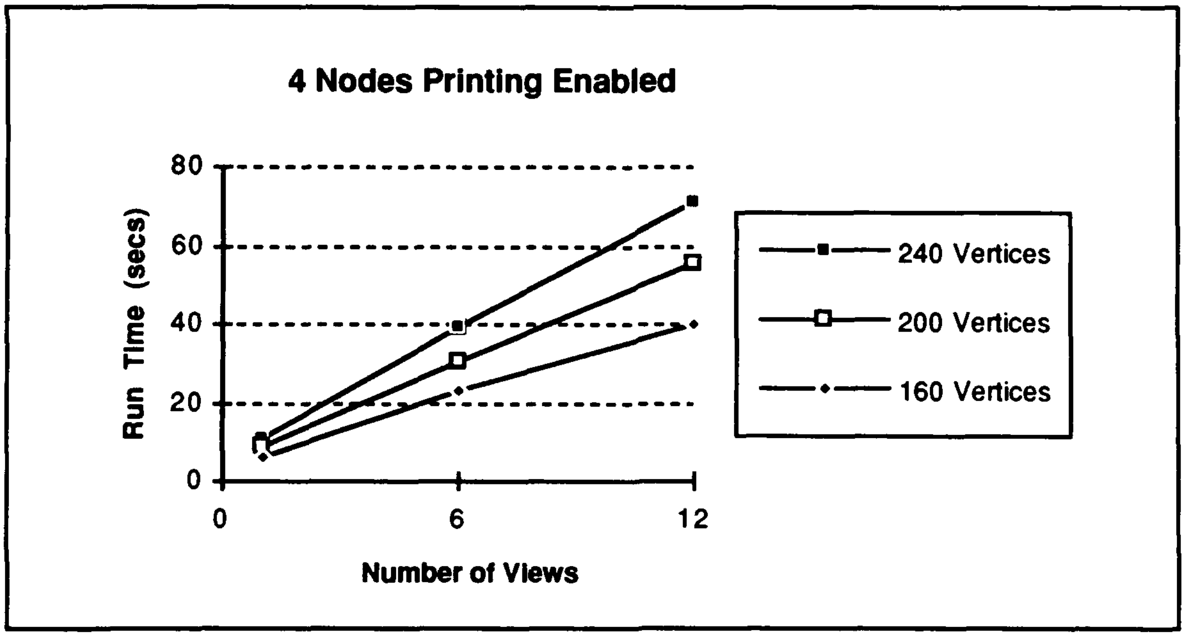

Figure 15. Four Nodes Printing Enabled

Figures 16 through 18 show the results when printing is enabled and the number of nodes is varied for 160, 200, and 240 vertices, respectively. In each case, the start object is assigned to node zero. The number of big objects that are created depends on the number of nodes used. With n nodes n-1 big objects are created, each of which performs approximately one n-th of the computation. The graphs show a characteristic shape regardless of the number of vertices. Performance for twelve views is clearly best with four nodes. As more nodes are used for the computation, performance is worse.

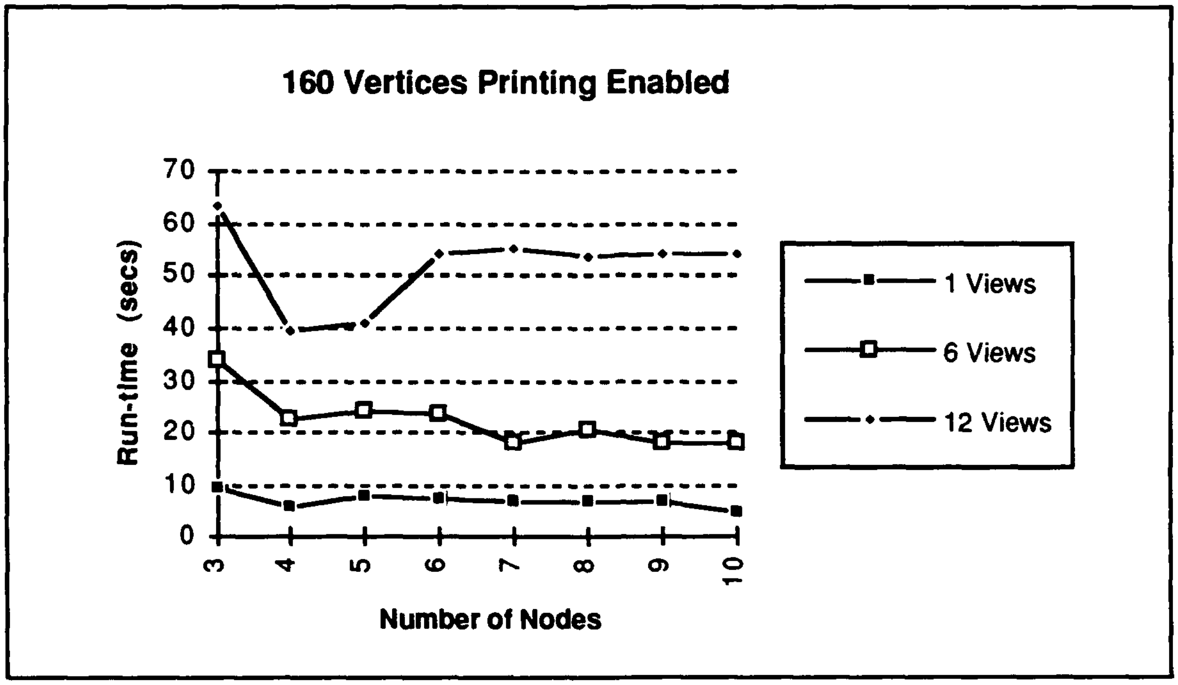

Figure 16. 160 Vertices Printing Enabled

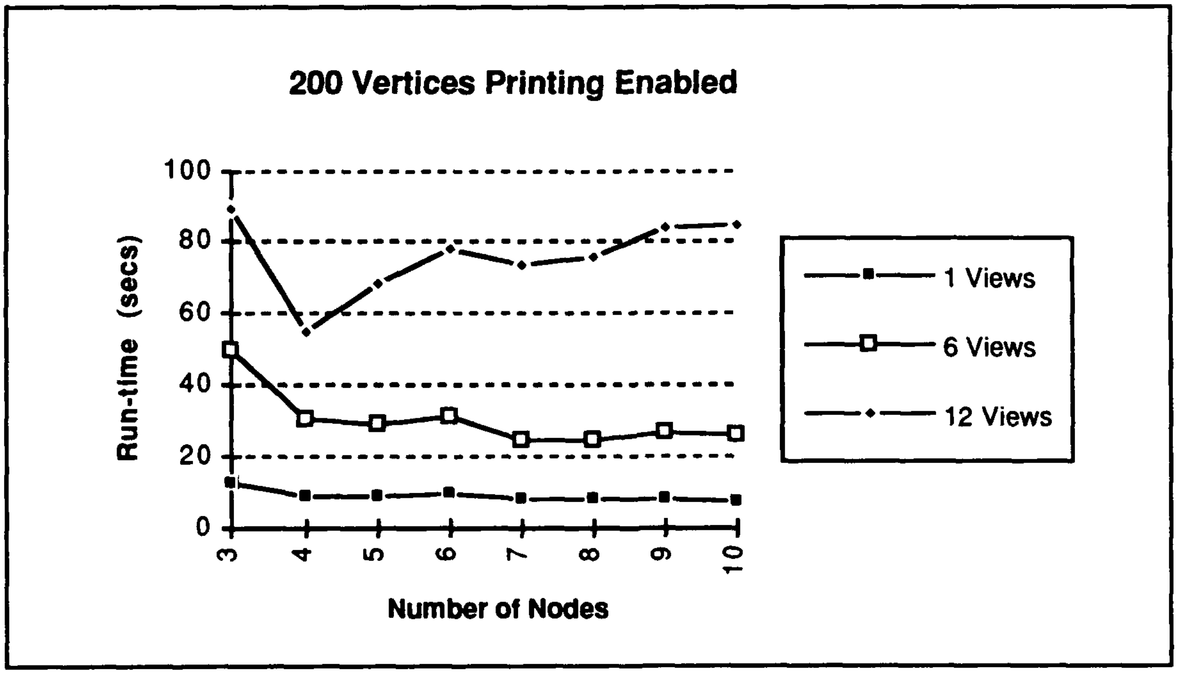

Figure 17. 200 Vertices Printing Enabled

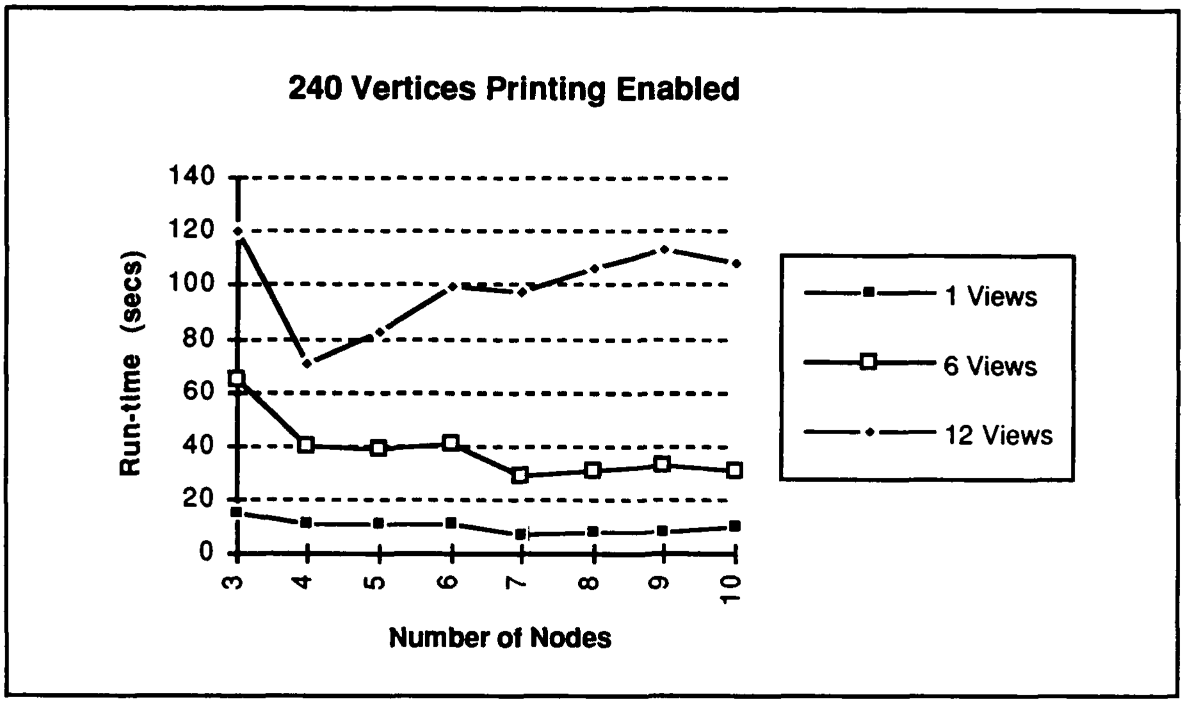

Figure 18. 240 Vertices Printing Enabled

When printing is suppressed in the application code, a different characteristic shape results (figure 19). The graphs for 160 and 240 vertices are similar. Performance improves as more nodes are used for the computation. The same pattern is preserved regardless of the number of vertices used in the test.

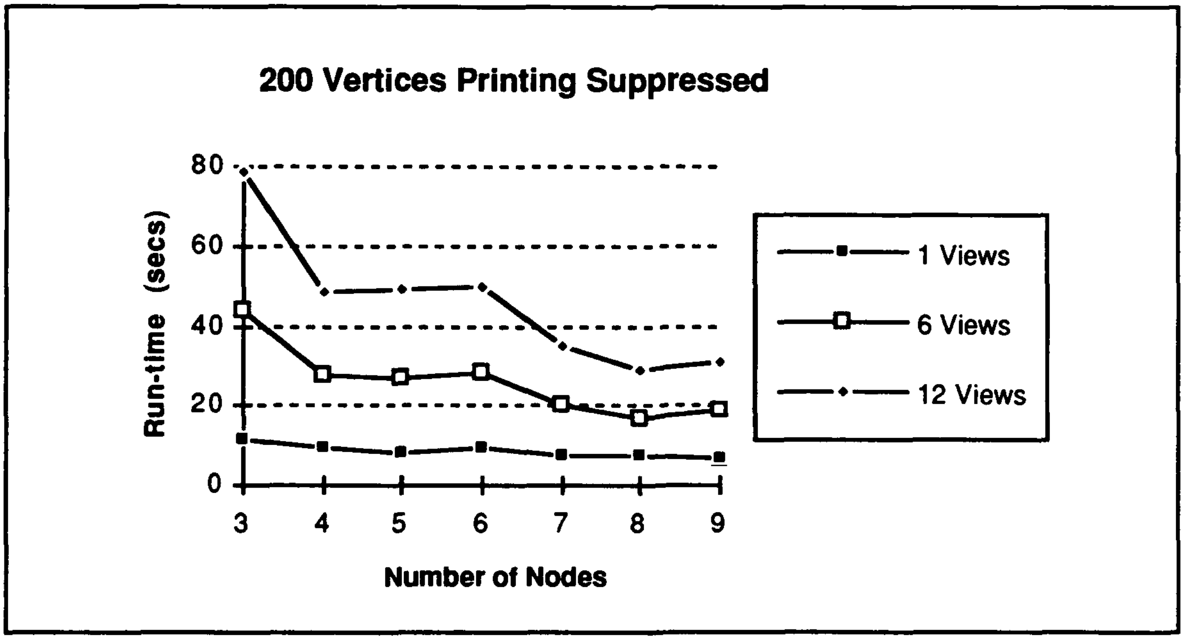

Figure 19. 200 Vertices Printing Suppressed

An interesting phenomenon may be observed when we examine the graphs that show how the performance varies when the number of views is varied on seven nodes. With printing suppressed, the run-time increases linearly with the number of views regardless of the number of vertices (figure 20).

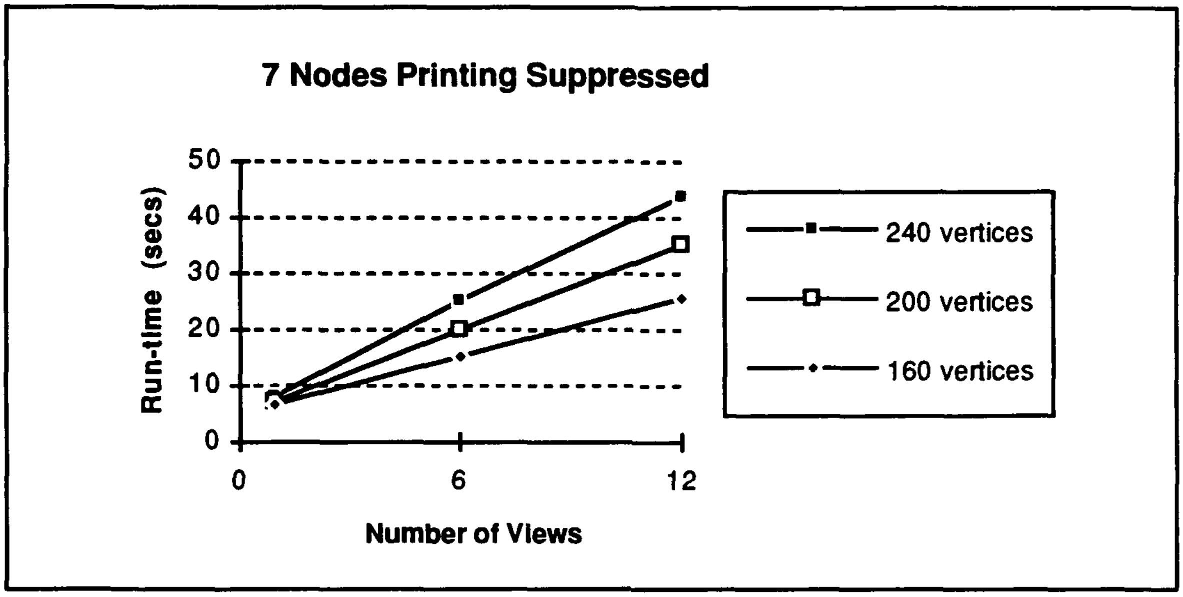

Figure 20. Seven Nodes Printing Suppressed

When printing is enabled using four nodes, figure 11 shows run-time increased linearly with the number of views. The same relationship holds for three nodes. However, with seven nodes run-time does not increase linearly with the number of views, as figure 21 indicates. Similar graphs can be drawn for five, six, seven, eight, nine, and ten nodes.

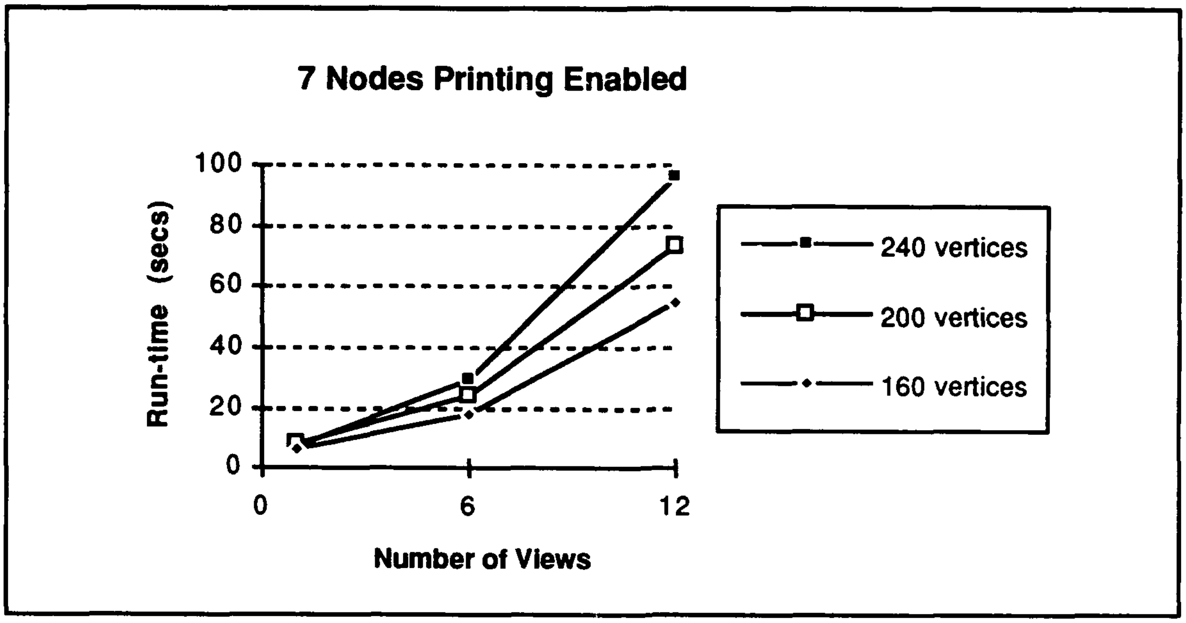

Figure 21. Seven Nodes Printing Enabled

Figure 22 shows a comparison of 200 vertices with printing suppressed for four nodes and seven nodes.

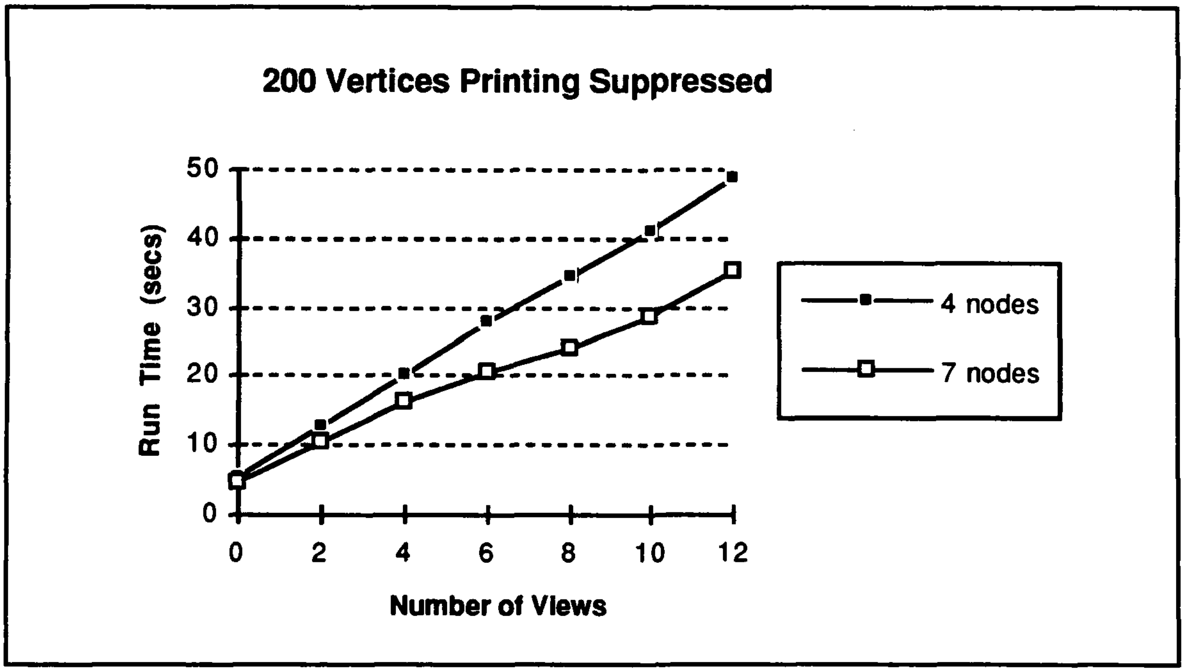

Figure 22. 200 Vertices Printing Suppressed Four and Seven Nodes

Figure 23 shows a comparison of 200 vertices with printing enabled for four nodes and seven nodes.

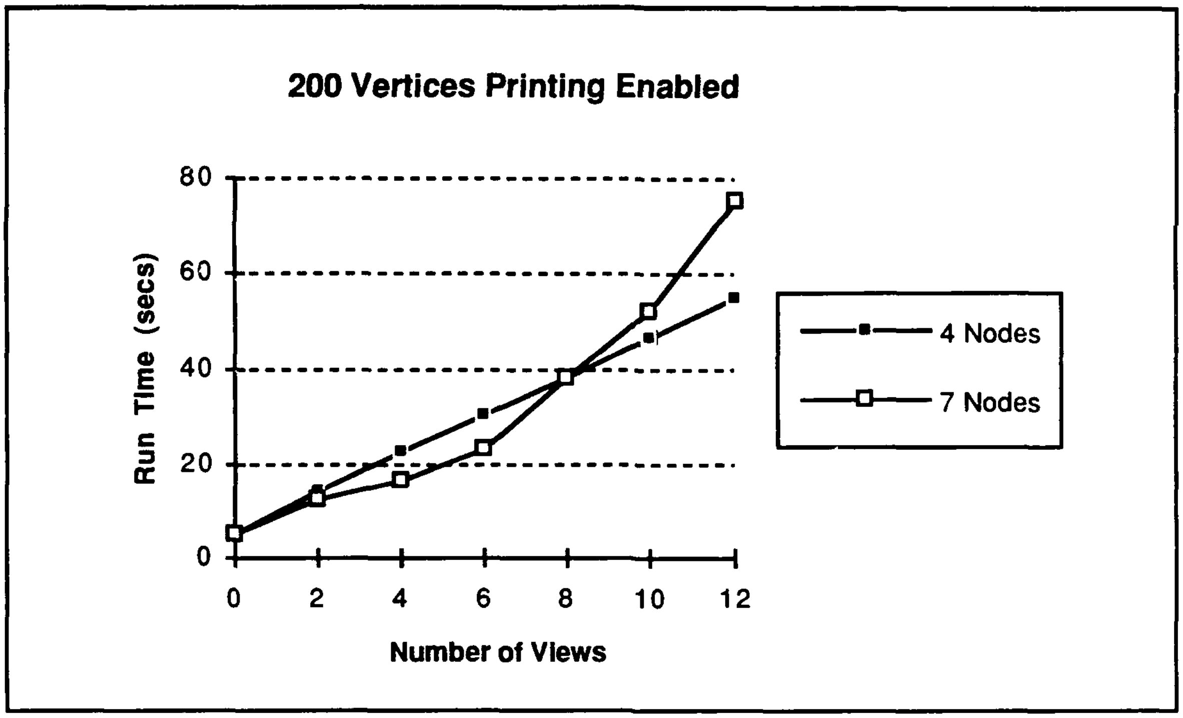

Figure 23. 200 Vertices Printing Enabled Four and Seven Nodes

For the remainder of this section we will explore various possible explanations for the non-linearly increasing run-time with the number of views when printing is enabled and more than four nodes are used. We will consider whether the output object could be a bottleneck or whether printing itself could be responsible. We will explore the possibility that more work is being done in the seven node implementation than in the four node implementation.

For each set of views, the number of statements printed is the same whether the application is run on four or seven nodes. Further, in SaM all output related messages are sent to the output object on node zero, regardless of how many nodes participate in the computation. It seems clear that the output object cannot be causing the non-linear run-time behavior since it processes the same number of messages for a set of views regardless of the number of other nodes.

We thought the cause of the non-linear behavior might be the actual printing itself. So we modified the output object so that it processed all print messages except for the actual printing of them. As figure 24 shows, printing in and of itself does not explain the non-linear behavior because the characteristic shape of the graph is similar (figure 17) when the statements are actually printed and when they are not (printing suppressed at the output object). Similar results were obtained for both 160 and 240 vertices.

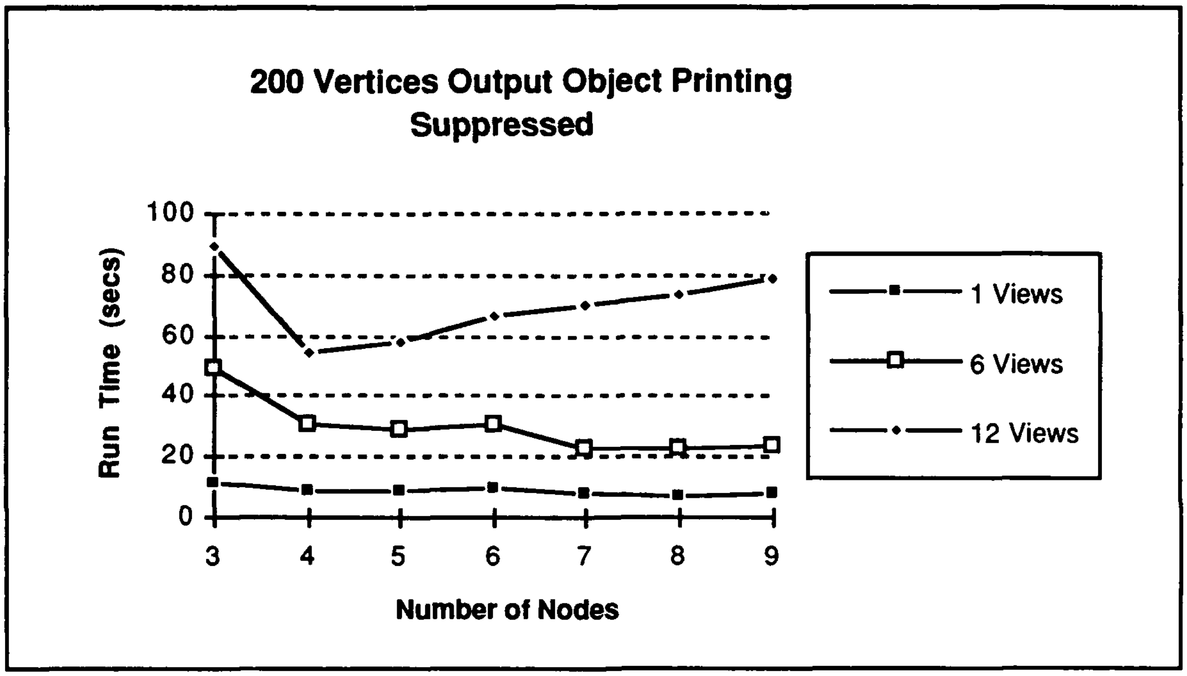

Figure 24. 200 Vertices Output Object Printing Suppressed

Figure 25 compares four and seven node executions with output object printing suppressed. We observe similar non-linear behavior as when the output object actually performs the printing.

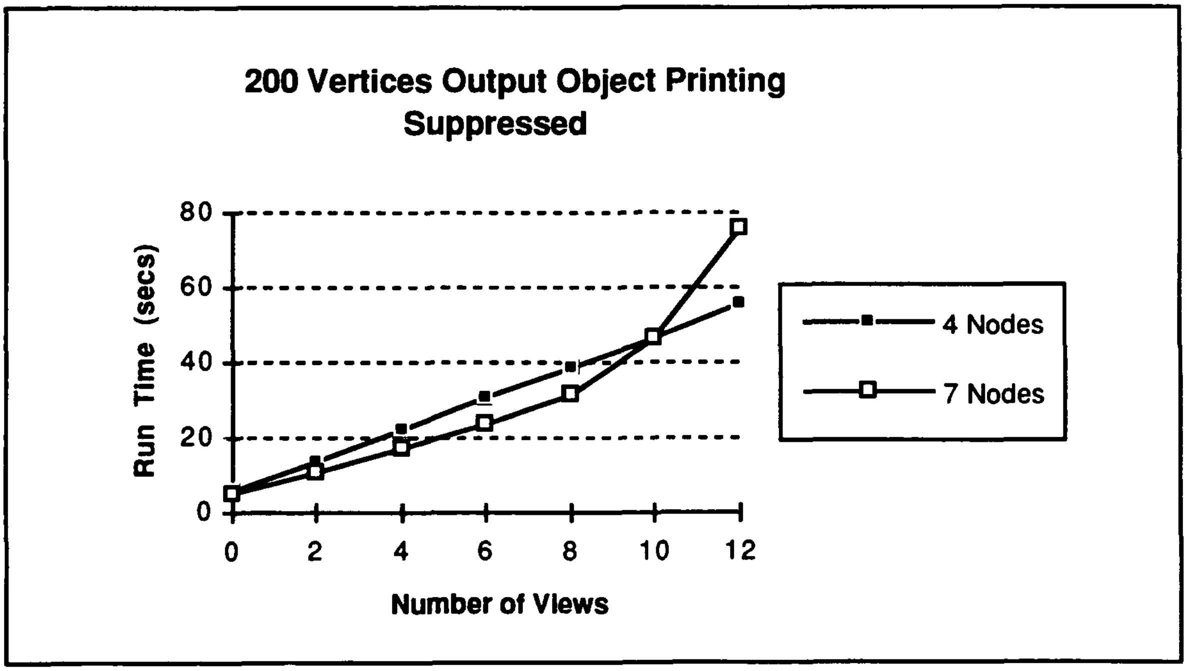

Figure 25. 200 Vertices Output Object Printing Suppressed Four and Seven Nodes

Might it be the case that the seven node implementation is doing more work than the four node implementation as the number of views increases? Since the application work is fairly evenly divided among the nodes (other than node zero), the four node implementation should do more application work per node than the seven node implementation. Node zero does somewhat more application work in the seven node case because it has to initiate the creation of more big objects and send more messages for each view angle set. However, the fact that when printing is suppressed at the application objects the run-time increases linearly with the number of views suggests that the work on the non-zero nodes is relatively well distributed and that the impact of the extra work performed by node zero is negligible. If the application work is evenly distributed, then the generation of the print messages is probably relatively evenly divided among three nodes in the four node case and among six nodes in the seven node case.

Might it be the case that the seven node implementation is doing more SaM work than the four node implementation as the number of views increases? To investigate this possibility, let us examine in detail the data collected during a SaM execution using four nodes and seven nodes. As we will see, table 3 indicates that indeed this seems to be the case. But is the amount of work increasing non-linearly with the number of view s?

Table 3. Data Collected During SaM Image Processing Runs

Nodes	4	7	4	7	4	7
Views	0	0	6	6	12	12
Vertices	200	200	200	200	200	200
Sample	1	2	3	3	2	3
Big objects	3	6	3	6	3	6
Application objects	4	7	4	7	4	7
Contexts	31	61	49	97	67	133
Futures	0	0	0	0	0	0
Multifutures	1	1	1	1	1	1
States saved	210	405	1,134	1,437	2,111	2,522
Instance variables	35	68	53	104	71	140
Environments	127	247	268	471	413	700
External messages	75	150	1,023	1,212	2,000	2,759
Internal messages	79	139	161	253	235	439
GVTs	3	3	13	12	21	39
GVT interval	1,000	1,000	1,000	1,000	1,000	1,000
Positive rollbacks	0	0	0	0	0	0
Negative rollbacks	0	0	0	0	0	0
States rolled back	0	0	0	0	0	0
Run-time (secs)	4.975	4.609	30.602	23.589	55.165	76.123

Before we attempt to answer that question, we need to have an understanding of what this data represents. *Nodes* is the number of S2010 nodes used: one for each big object and one for the start object. *Views* is the number of different view angle sets processed for the image. *Vertices* is the number of vertices in the image; it provides an indication of the complexity of the image. *Sample* is the test case used. Each case was run three times. The table shows the sample test case that produced the middle run-time of the three executions. The differences between run-times for the test cases was observed to be negligible. *Big objects* is the number of big objects that are used to process a view of the image. Each big object computes the triangles for each view set and then processes a portion of the lines to determine whether they are hidden or partially hidden by any one of the triangles. *Application objects* is the number of concurrent application objects created dynamically during the execution of the program: one for each big object and one for the start object. *Contexts* is the number of context objects created dynamically during the program execution. It is a measure of the number of application messages processed that are not explicitly send to self. As figure 8 indicates, each view angle set involves processing one application message per big object. In addition, 10 messages per big object are used to initialize each big object for an image. These messages contain the coordinates of the vertices and descriptions of the polygons in terms of indices of their vertices. Thus, each big object processes 10 messages for zero view angle sets, 16 for six view angle sets, and 22 for 12 view angle sets. The start object processes one message. *Futures* is the number of futures created: no futures were used in this application. *Multifutures* is the number of multifutures created: one to create all the big objects.

States saved is the number of states saved by application objects, contexts, futures, multifutures, and creators. A state contains the information needed by the synchronization manager to control rollback. A state may contain a copy of the current value of the instance variables for application objects or environment variables for context objects. In the case of application objects, futures, multifutures, and creators, a state contains an input message and the output message that was sent in response (if appropriate). For contexts, the state may contain an input message, for example, a reply from a future. However, contexts also save a state when a message is sent as a result of method execution. *Instance variables* is the number of copies of the application instance variables saved for each application object. *Environments* is the number of copies of the application instance variables, methods local variables, and arguments saved for each context object. A state for an application object, future, or multifuture may or may not contain a copy of the application instance variables. For example, a future need only have one copy of the value it has been set for. Messages that are processed to get that value need not have individual copies of that value. A state for a context may or may not contain a copy of the environment. For example, a state is saved each time a print message is sent, but it is not necessary to save a copy of the environment each time because a print message can never cause a rollback due to a recursive cycle. On the other hand, an application request message can cause a rollback due to processing a recursive cycle, so a copy of the environment must be saved.

External messages is the number of messages sent between objects on different processors. *Internal messages* is the number of messages sent between objects on the same processor. External and internal messages include application messages as well as GVT messages, messages between application objects and contexts, and messages between contexts and futures or multifutures. *GVTs* is the number of GVT computations performed during the computation. *GVT interval* is the minimum amount of time that must pass before a new GVT calculation can begin during normal computation. When the only object that reports an OVT of less than infinity is the output object, GVT calculations occur more frequently. The GVT interval we used for this application was a thousand milliseconds, or one second. *Positive rollbacks* is the number of rollbacks that occurred during the computation due to messages being processed out of order. *Negative rollbacks* is the number of rollbacks performed as a result of negative messages received (a negative message is a message that must be retracted because another message was processed out of order). *States rolled back* is the number of states that were rolled back during the computation. Run-time indicates the execution run-time in seconds from the time the harness on node 0 is initialized to the time when the computation is complete, including all output, that is, the time when GVT is determined to be infinity.

As we can see from the table, more work is being done in the seven node implementation than in the four node implementation: more contexts are being created; more states, instance variables, and environments are being saved; more messages are being sent, both external and internal; and for twelve views, more GVT computations are being performed.

We shall look at the data collected on individual nodes to see if we can narrow our investigation further. Table 4 shows the data collected on node 1. Although the types of data collected are similar to that shown in table 3, in table 4 they indicate the activity on only node one.

Table 4. Data Collected During SaM Image Processing Runs for Node 1

Nodes	1 of 4	1 of 7	1 of 4	1 of 7	1 of 4	1 of 7
Views	0	0	6	6	12	12
Vertices	200	200	200	200	200	200
Sample	1	2	3	3	2	3
Big objects	1	1	1	1	1	1
Application objects	1	1	1	1	1	1
Contexts	10	10	16	16	22	22
Futures	0	0	0	0	0	0
Multifutures	0	0	0	0	0	0
States saved	53	53	354	192	662	338
Instance variables	11	11	17	17	23	23
Environments	30	30	70	59	109	89
External messages	4	4	285	122	571	265
Internal messages	20	20	32	32	44	44
Positive rollbacks	0	0	0	0	0	0
Negative rollbacks	0	0	0	0	0	0
States rolled back	0	0	0	0	0	0

The data collected on nodes other than node zero is similar to the data collected on node one. Clearly the execution with four nodes is doing both more SaM and more application work than the seven node executions on nodes other than node zero.

Figures 26 through 28 show graphs that indicate that the number of states saved, number of environments saved, and number of external messages sent (including those that cause lines to be printed) on nodes other than node zero. As the figures show these parameters increase linearly with both four and seven nodes. In addition, they show that in the four node executions the amount of SaM work increases more rapidly with the number of views than the seven node executions.

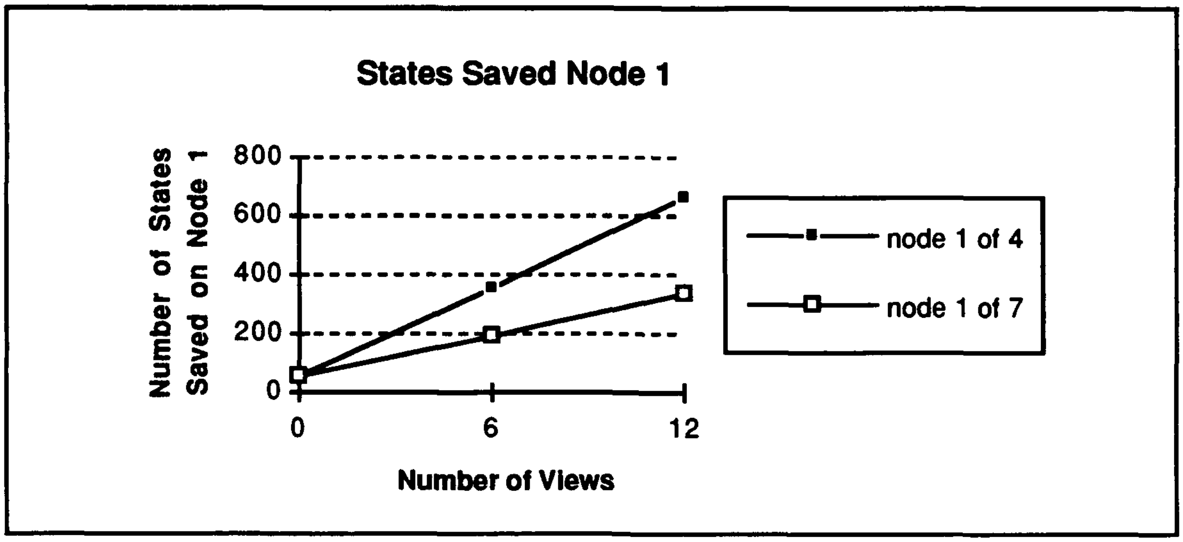

Figure 26. States Saved on Node 1

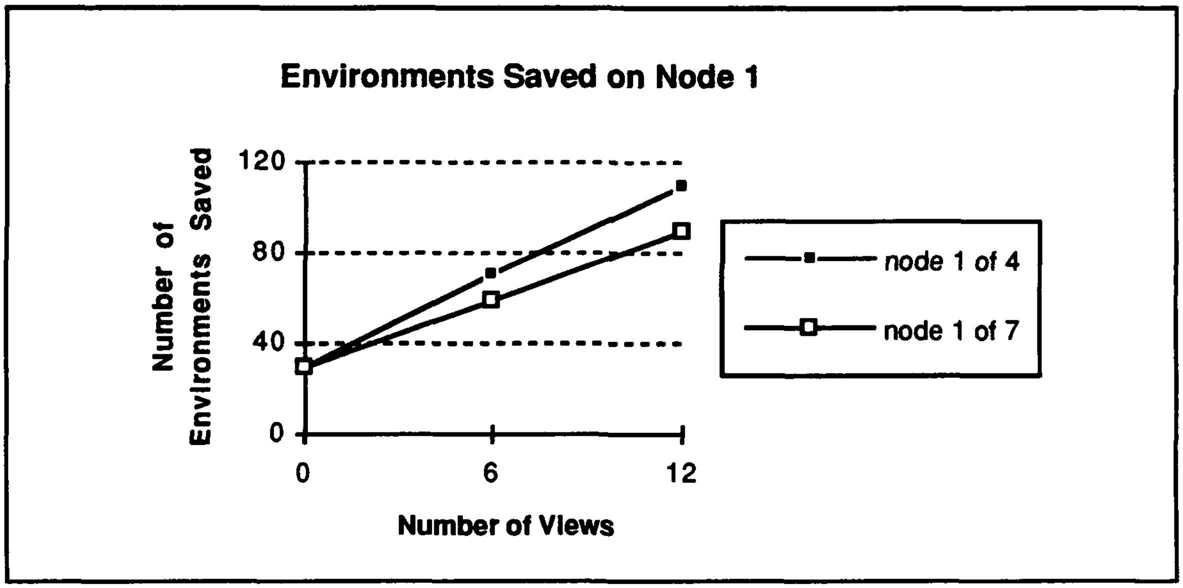

Figure 27. Environments Saved on Node 1

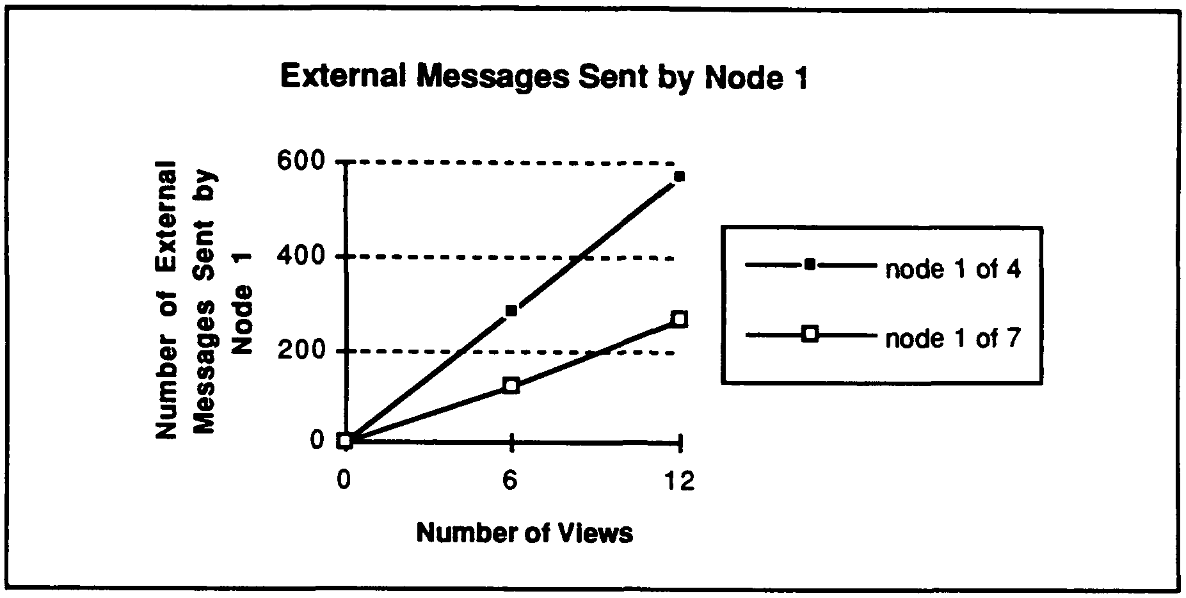

Figure 28. External Messages Sent by Node 1

If the nodes other than node zero are not responsible for the non-linear behavior, perhaps node zero is responsible. Table 5 shows the same data for node zero.

Table 5. Data Collected During SaM Image Processing Runs for Node 0

Nodes	0 of 4	0 of 7	0 of 4	0 of 7	0 of 4	0 of 7
Views	0	0	6	6	12	12
Vertices	200	200	200	200	200	200
Sample	2	3	3	3	2	3
Big objects	0	0	0	0	0	0
Application objects	1	1	1	1	1	1
Contexts	1	1	1	1	1	1
Futures	0	0	0	0	0	0
Multifutures	1	1	1	1	1	1
States saved	51	87	75	129	99	171
Instance variables	2	2	2	2	2	2
Environments	37	67	61	109	85	151
External messages	63	126	171	324	261	846
Internal messages	19	19	65	61	103	175
Positive rollbacks	0	0	0	0	0	0
Negative rollbacks	0	0	0	0	0	0
States rolled back	0	0	0	0	0	0

As the table indicates, more SaM work is being done by node zero in the seven node execution than in the four node execution. Clearly the execution with seven nodes is doing both more SaM and more application work than the four node executions on node zero.

Figures 29 through 31 show graphs that indicate that the number of states saved, number of environments saved, and number of external messages sent on node zero increase with both four and seven nodes. In addition, they show that in the seven node executions the amount of SaM work increases more rapidly with the number of views than the four node executions. We note that the number of states saved and the number of environments saved in both the seven node and four node executions increase linearly with the number of views. However, while in the four node execution the number of external messages sent by node zero increases linearly with the number of views, in the seven node case the increase is non-linear.

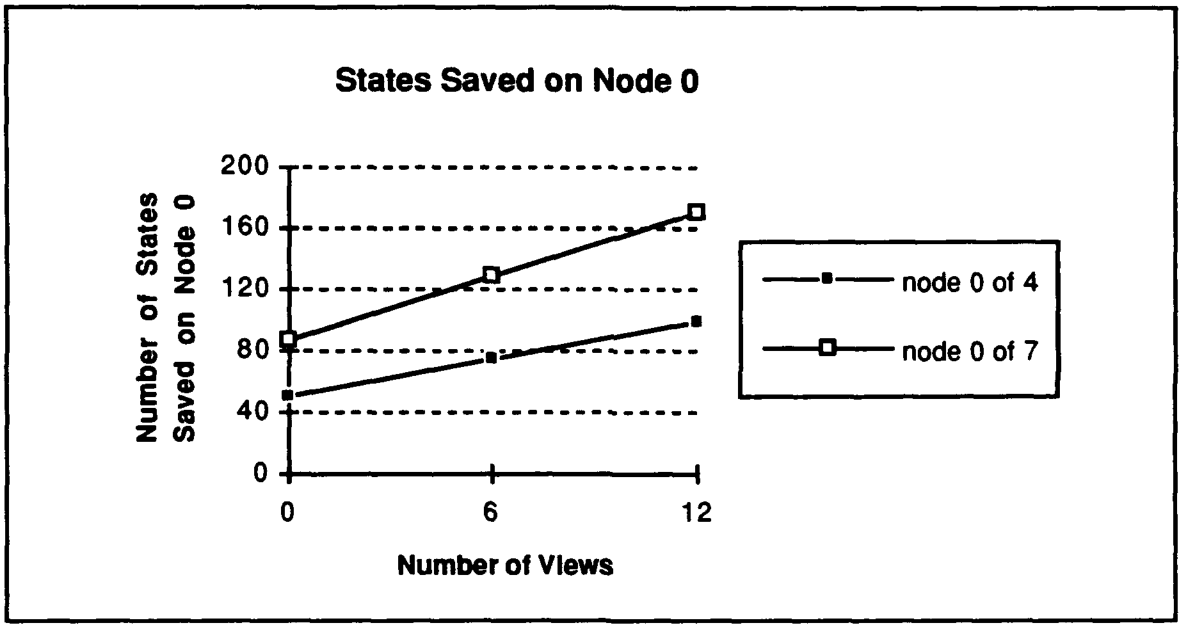

Figure 29. States Saved on Node 0

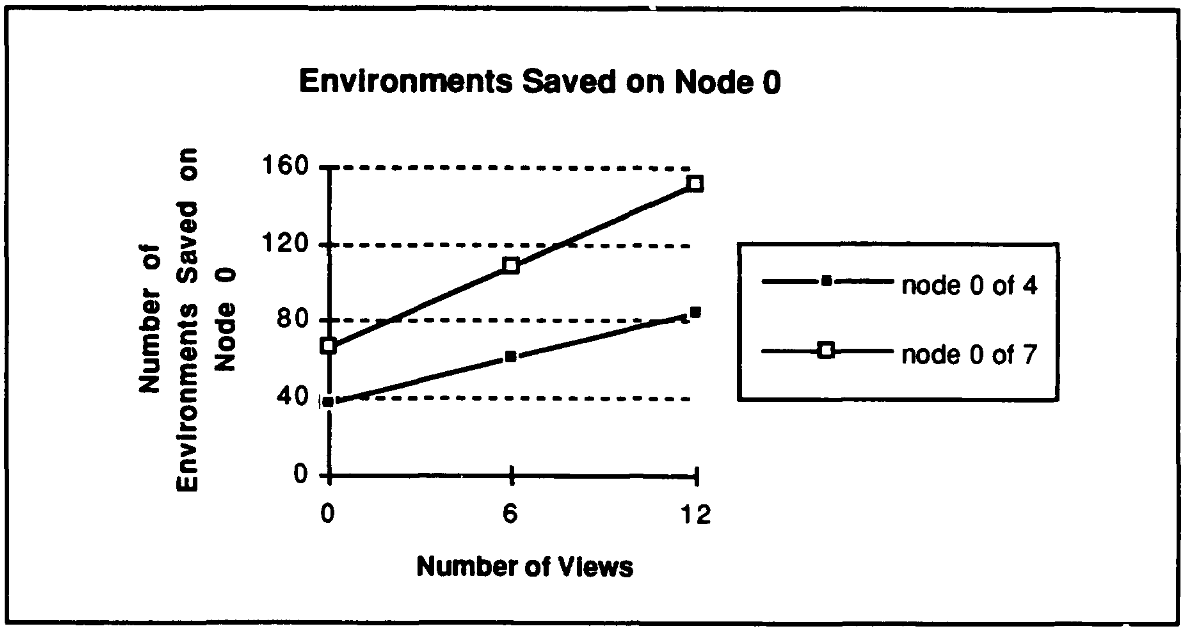

Figure 30. Environments Saved on Node 0

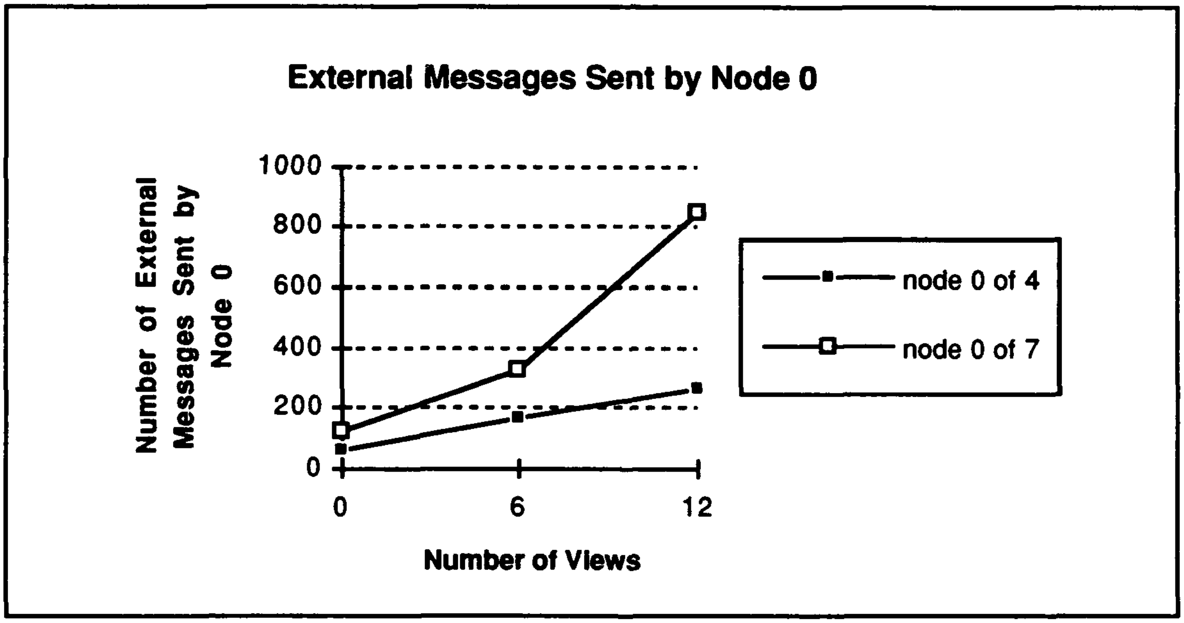

Figure 31. External Messages Sent by Node 0

We now turn our attention to the cause of these additional messages sent by node zero. It is true that in the seven node executions node zero sends more application messages to initiate the creation of more big objects and to send each big object the view angle sets. In this application, the number of contexts created is the same as the number of application messages sent by node zero. (In other applications we shall consider in this paper, this will not be the case.) As figure 32 indicates, the number of application messages sent by node zero increases more rapidly with the number of views in the seven node execution than in the four node execution. However, the increase is linear. The number of messages sent by node zero to create the big objects on other nodes, although larger in the case of seven nodes (six as opposed to three), is independent of the number of views.

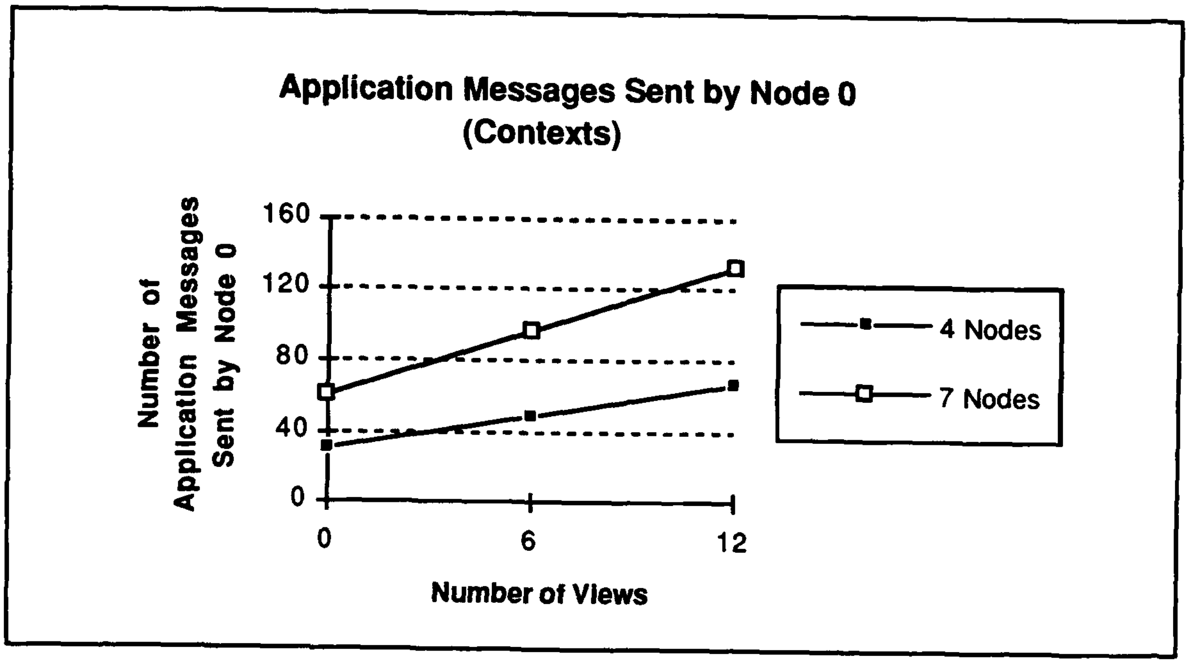

Figure 32. Application Messages Sent by Node 0

Where are the additional external messages coming from? In addition to the start object and the output object, the GVT master is also located on node zero. GVT computations are basically performed according to the real-time clock. Since the run-time in the seven node executions increases non-linearly with the number of views, the number of GVT computations performed increases non-linearly with the number of views as figure 33 indicates.

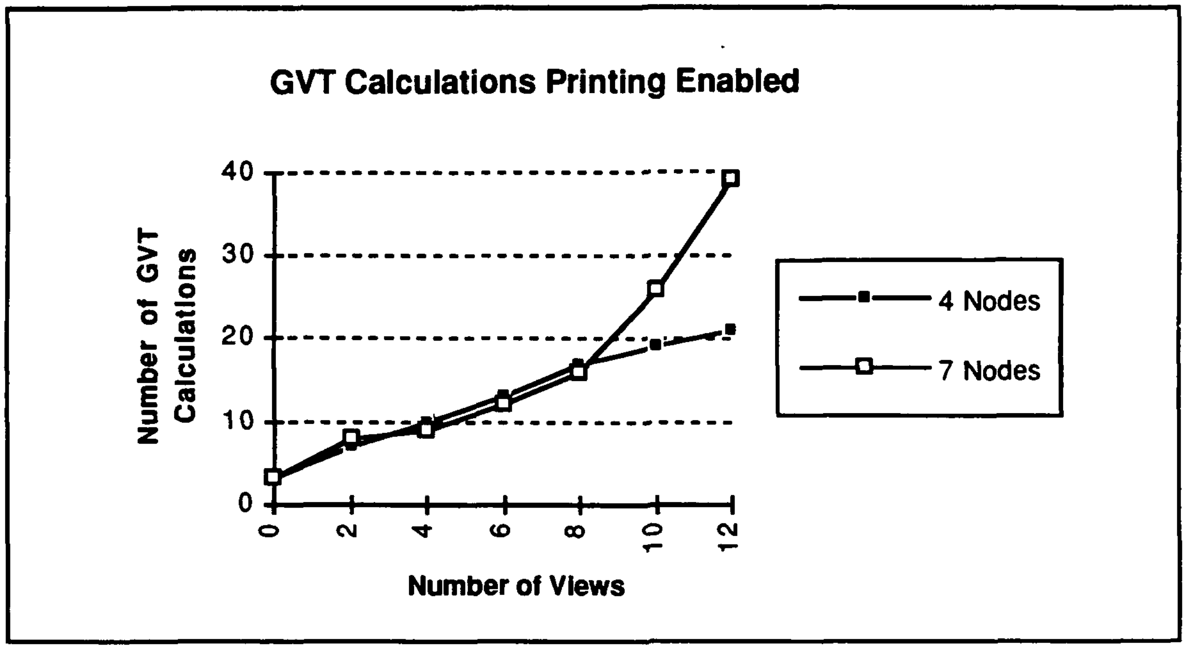

Figure 33. GVT Calculations Printing Enabled

In the seven node implementations, each GVT calculation involves node zero sending six external messages to tell the other nodes to prepare for a GVT calculation, six external messages to tell the other nodes to perform an OVT calculation, and six external messages to tell the other nodes the new GVT assignment. Further, node zero performs more work to accomplish a GVT calculation with seven nodes as it must process six messages to determine the new GVT value. Since the number of GVTs is proportional to run-time, however, what we are seeing here is an effect, not a cause, of the non-linear behavior.

Since we could not identify how SaM might be responsible for this non-linear behavior, we began to look elsewhere for an explanation. The Symult S2010 is a mesh architecture. Messages in the four node executions have less far to travel to reach node zero on average and are less likely to encounter contention than messages in the seven node executions. To investigate the possibility that message contention could be responsible for the non-linear behavior, we wrote a small program that initiates a simple computation on a number of nodes—SaM is not used. The computation involves each node, other than node zero, preparing a number of messages of the same type as those sent by the big objects to node zero. These messages are short, less than 100 bytes. When node zero receives these messages, it prints their contents. Figures 34 and 35 show the results of this experiment. In figure 34, each plotted point indicates the average run-times of one hundred executions of the same program. In figure 35 each plotted point indicates the difference of the maximum and minimum run-times observed during these tests.

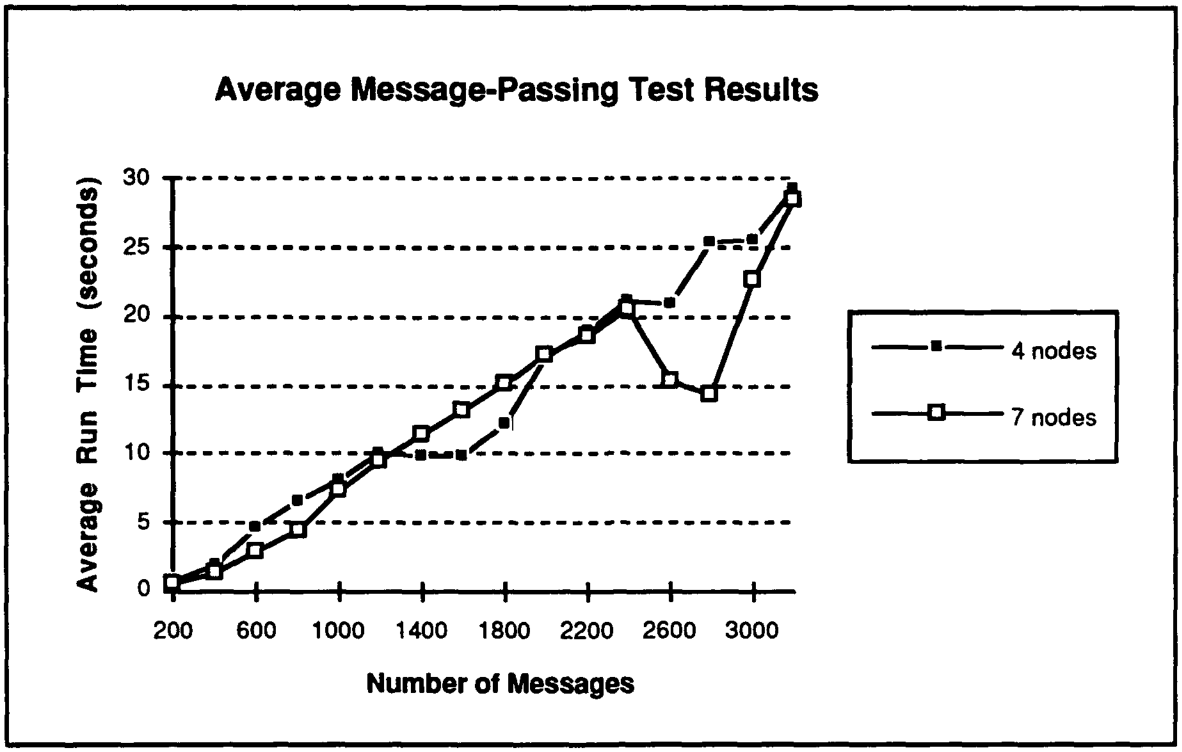

Figure 34. Send Messages and Print Only

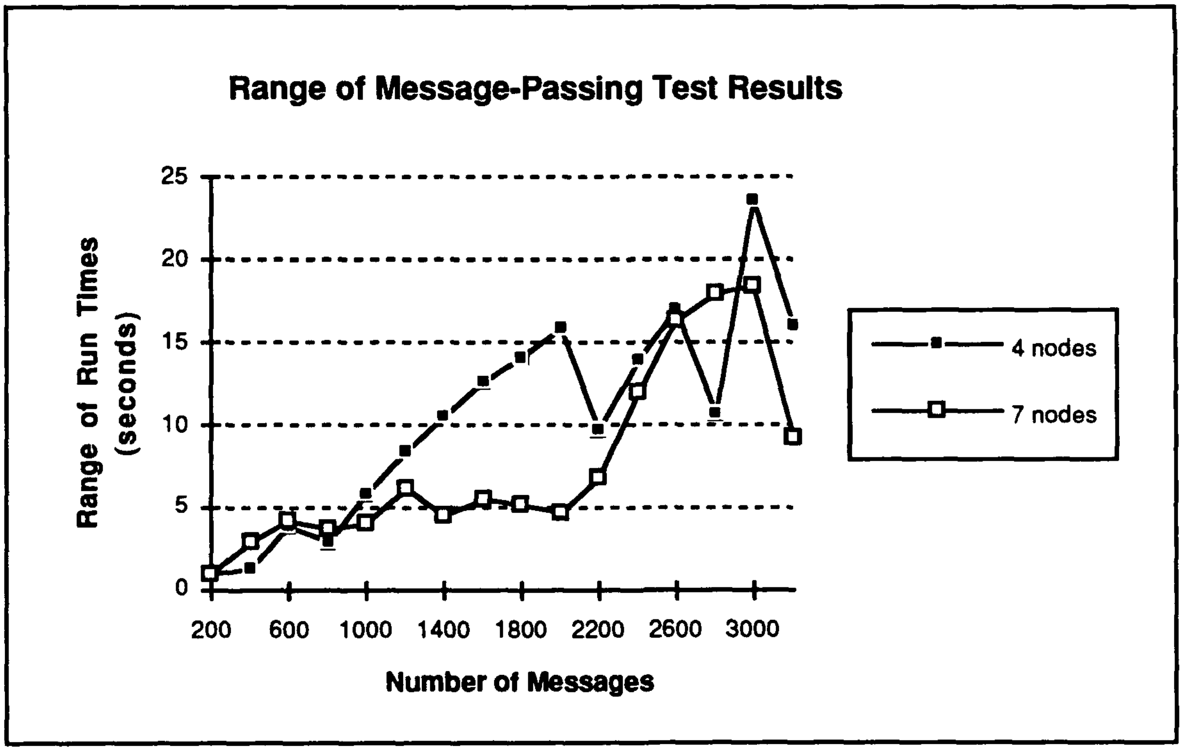

Figure 35. Send Messages and Print Run-Time Ranges

We can see that larger number of messages make an execution more unstable with respect to run-time, but we see no significant distinction between our four node results and our seven node results. At this point we are unable to identify the cause of the non-linear behavior.

SECTION 6

NEURAL NETWORK APPLICATION

We implemented an initial prototype of a neural network training application that employs a modified version of the backpropagation algorithm found in [McClelland:88]. The program reads backpropagation control parameters and the network description from files. Each network has three or more layers: one input layer, one or more hidden layers, and one output layer. The units in each layer are fully connected to the units in the next layer (connections do not skip between layers). Each input connection has an associated weight that represents that connection's contribution to the activation value of the unit to which it is connected. Figure 36 shows a network that can identify whether a six-bit pattern is symmetric or not. An input consists of six-bits, for example, 010011 or 010010, and a one-bit output 0 for false and 1 for true.

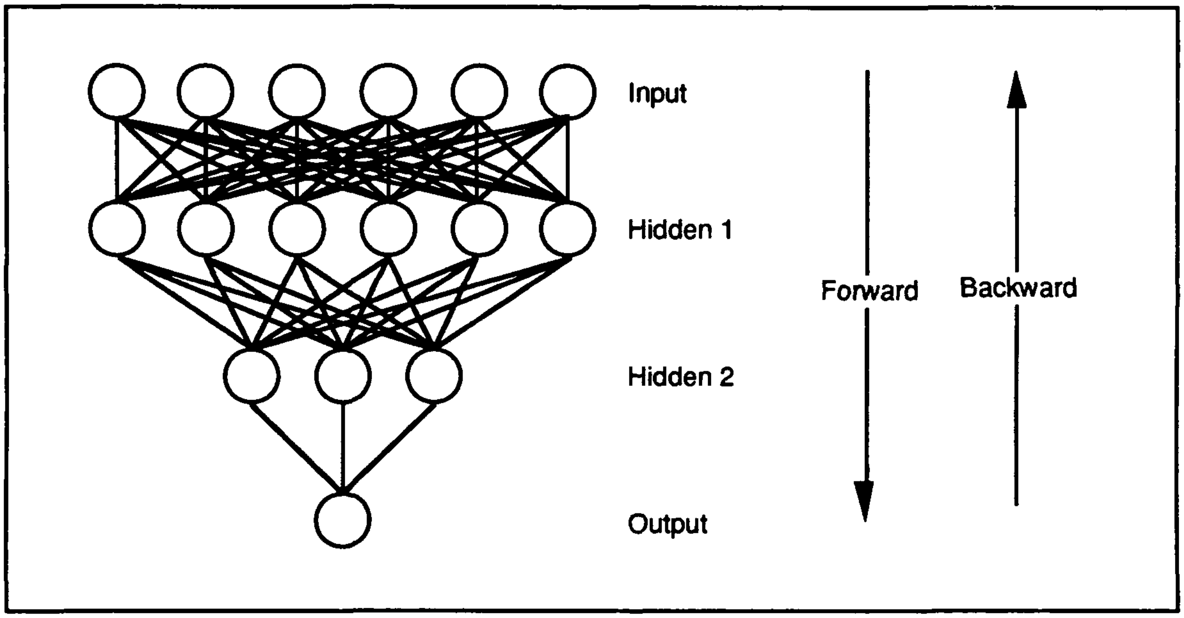

Figure 36. Neural Network for Symmetry

The backpropagation algorithm consists of computing the output values of the network (forward propagation), computing errors based on the difference between the target and actual output values, then backpropagating these error terms through the network in order to change the weights at each unit connection. The activation value of each unit is (some function of) the weighted sum of all its input connections.

The basic modification made to the backpropagation algorithm changed the control flow in order to make the algorithm more amenable to parallelization. The original algorithm performs the steps mentioned above in an inherently sequential fashion. In the modified algorithm, each unit expects a known number of activation values from all its input connections. The unit acts like a collector, calculating its activation value incrementally based on input it receives from its connections. When the unit has received an activation value from all its input connections, it can calculate its own activation value and pass that value to the units connected to its output connections. Thus, activation values are transmitted from the input layer, to the hidden unit layer, then to the outputs where the error term can be calculated. The units also act as collectors for calculating and backpropagating the error term for altering the weights on each connection.

A training cycle consists of calculating activation values and backpropagating the error for all input patterns. This training cycle represents one epoch. Backpropagation is performed for a maximum number of epochs or may stop before the maximum number of epochs is reached if the error at the output units for all patterns becomes small enough.

In our original implementation, each unit was a concurrent object. The granularities of the computations (number of instructions executed per communication event) were too fine compared to the overhead incurred by communication using SaM. We modified the neural network application to contain more coarse-grained parallelism than was present in the first implementation. We used a technique for parallelizing neural network simulations found in [Pomerleau:88]. Instead of representing individual units in a single network as concurrent objects, this implementation replicates the network and divides the training patterns evenly among the resulting networks. These networks are the objects that execute in parallel and provide for more coarse grained parallelism. Our algorithm is very similar to the one presented in the paper; however, our algorithm is more generic while theirs was more tailored to the parallel machine on which it was implemented, the Carnegie Mellon University Warp, a ten processor, programmable systolic-array computer.

The multiple network model must be trained somewhat differently than the single network model. In the single network simulation, the network calculates the output for all patterns, and subsequently calculates the error signal which it back propagates through the network changing the input weights at each layer.

In the multiple network model, each network calculates the outputs for its own set of patterns, back propagates the error, and determines what the weight change would be, but does not update the weights as yet. Before continuing to the next epoch, each network sends their respective weight changes to a controller, called a weight-summer, that sums all the weight changes and broadcasts to all networks the new weight values for all of the input weights to each layer. The start object controls the overall training. It tells each network when to begin a new training epoch. It queries the weight summer to determine if training is complete at the conclusion of each epoch. Figure 37 shows the control flow for one epoch. The N's are the networks; WS is the Weight Summer. The arrow from WS to the start object is the reply that indicates whether training should continue.

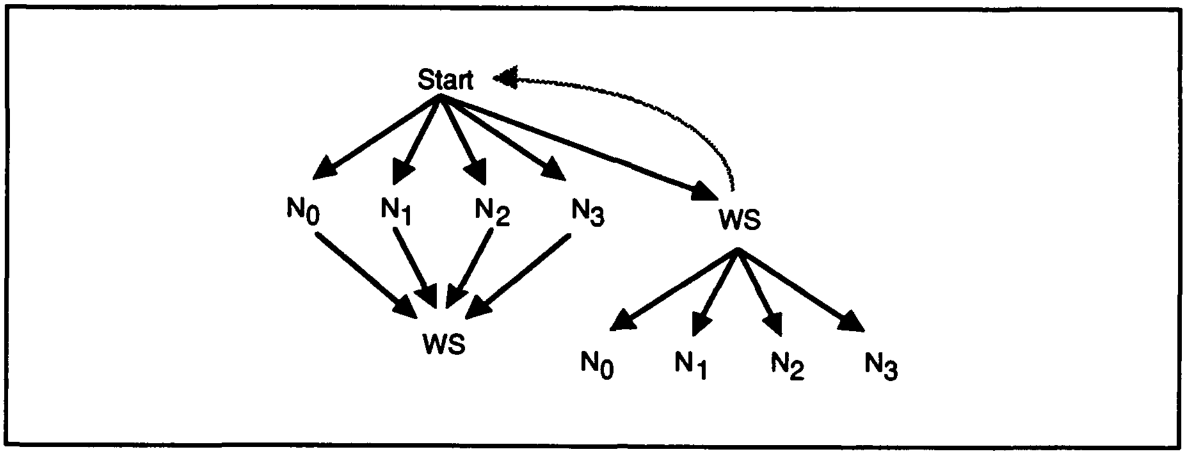

Figure 37. Neural Network Application Control Flow

There are obvious data dependencies between epochs, since updating the weights in a previous epoch affects the activation values calculated in the next epoch. Although SaM handles data dependencies correctly, it may do so by unnecessary roll back. Since the behavior of the backpropagation algorithm is well understood, we can place a synchronization barrier in the code to prevent the synchronization manager from spawning unnecessary parallel tasks that will be rolled back. A synchronization barrier is quite simple to add to the application code. We use a similar construct to an *assert* in the Lisp language that functions in a manner similar to a guard in the communicating sequential process model [Hoare:78].

The start object sends messages to the networks and the weight-summer in rapid succession. As the computation performed by the networks is relatively substantial, the weight-summer might receive and process the message from the start object before it has received all the messages from the networks. The weight-summer should not process the message from the start object until it has processed a message from each network. If the weight-summer processed this message, it would have to be rolled back when the network messages came in. Further, the start object might start a new training epoch at each network before weights had been updated by the weight-summer causing unnecessary rollbacks for the networks. Placing an assert on the message from the start object to the weight-summer that tests that the value of the weight-summer's instance variable inputReceived is equal to the number of networks before the message from the start object is processed ensures that this message will not be processed until all the messages from the networks have been received and processed (figure 38). When the message from the start message is processed, inputReceived is set to zero.

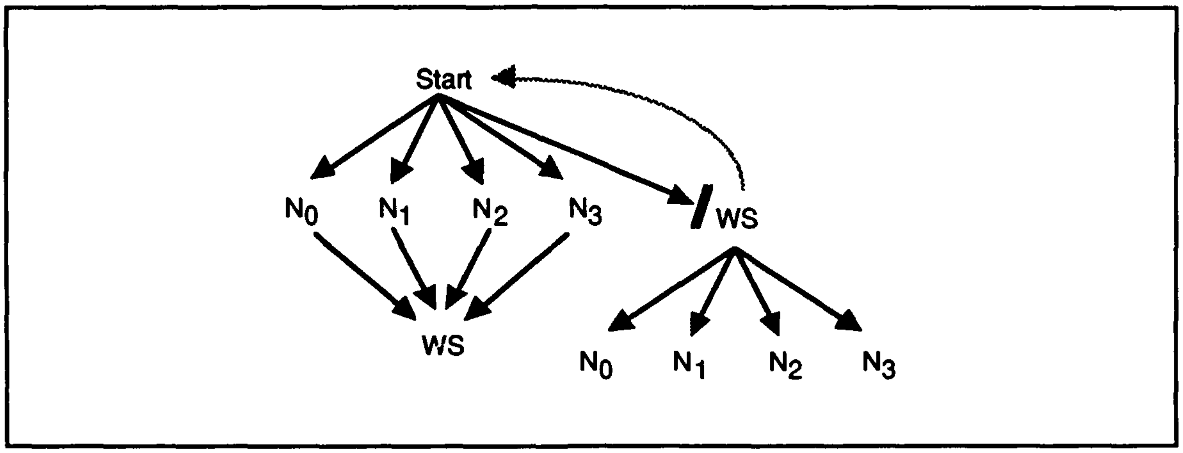

Figure 38. Neural Network Application Control Flow with Assert

We captured a number of parameters from the execution of training a network to identify whether a seven-bit pattern is symmetric or not. These are presented in table 6.

Table 6. Data Collected During a Training Session

Epochs	1,074
Hiddens	9
Patterns	128
Nets	4
Nodes	6
Application objects	6
Contexts	14,006
Futures	1
Multifutures	1,075
States saved	92,635
Instance variables	14,002
Environments	59,223
External messages	23,919
Internal messages	34,107
GVTs	442
GVT interval	1,000
Positive rollbacks	11
Negative rollbacks	8
States rolled back	36
Run-time (secs)	890.99

Epochs is the number of epochs required to train the network. *Hiddens* is the number of hidden units. *Patterns* is the number of input patterns. *Nets* is the number of networks among which the patterns are divided: each network is assigned 32 of the 128 patterns.

Nodes is the number of S2010 nodes used in the training: one for each net, one for the start object and one for the weight-summer. *Application objects* is the number of concurrent application objects created dynamically during the execution of the program: one for each net, one for the start object and one for the weight-summer. *Contexts* is the number of context objects created dynamically during the program execution. It is a measure of the number of application messages processed that are not explicitly send to self. As figure 38 shows, each epoch involves processing 13 application messages. Since there are 1074 epochs, this accounts for 13,962 contexts. Application messages to initialize the networks and the weight-summer and messages processed out of order that result in contexts created unnecessarily account for the other 44 contexts created. *Futures* is the number of futures created: one to create the weight-summer. *Multifutures* is the number of multifutures created: one to create the networks and one for each epoch. *States saved* is the number of states saved by application objects, contexts, futures, multifutures, and creators. *Instance variables* is the number of copies of the application instance variables saved for each application object. *Environments* is the number of copies of the application instance variables saved for each context object. An environment contains values of the application instance variables, the methods local variables and arguments.

External messages is the number of messages sent between objects on different processors. *Internal messages* is the number of messages sent between objects on the same processor. External and internal messages include application messages as well as GVT messages, messages between application objects and contexts, and messages between contexts and futures or multifutures. *GVTs* is the number of GVT computations performed during the computation. *Positive rollbacks* is the number of rollbacks that occurred during the computation due to messages being processed out of order. *Negative rollbacks* is the number of rollbacks performed as a result of negative messages received (a negative message is a message that must be retracted because another message was processed out of order). *States rolled back* is the number of states that were rolled back during the computation. Run-time indicates the execution run-time in seconds from the time the harness on node zero is initialized to the time when the computation is complete, including all output, that is when GVT is determined to be infinity.

Figure 39 shows the results of varying the number of networks in both the sequential and the SaM implementations. In the SaM implementations, since each network was assigned to a different node, the number of nodes were varied also. We were not able to obtain data for the two network (four node) case. We believe this was because of a lack of adequate application memory and/or stack space on the S2010 nodes.

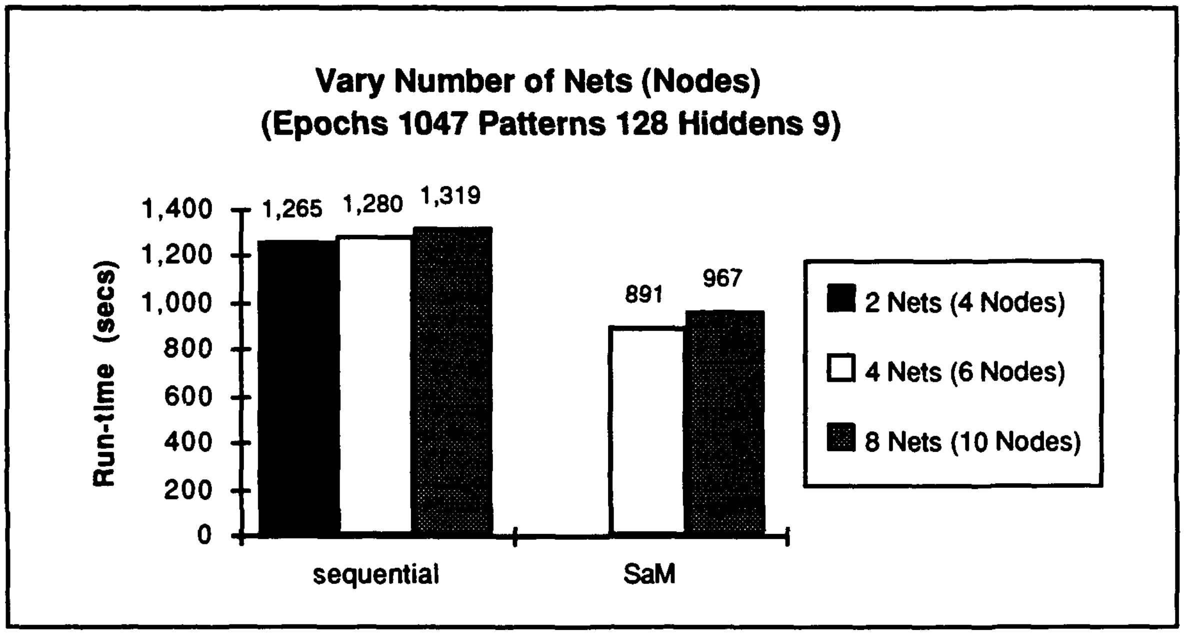

Figure 39. Vary Number of Nets

Figure 40 shows the results of the SaM and sequential executions when the number of patterns is varied. Five-bit symmetry has 32 patterns associated with it; six-bit has 64 patterns; seven-bit has 128 patterns. Five-bit symmetry trains in 189 epochs; six-bit symmetry trains in 431 epochs; seven-bit symmetry trains in 1074 epochs. As can be seen in the figure, the SaM execution performs better than the sequential execution for 64 and 128 patterns. Further, the run-time of the SaM execution is increasing linearly as the number of patterns increases but at a slower rate than the sequential execution (0.4 times the sequential rate of increase when printing is enabled in SaM but not in the sequential version). This is not surprising, since increasing the number of patterns increases the amount of computation that must be performed per epoch.

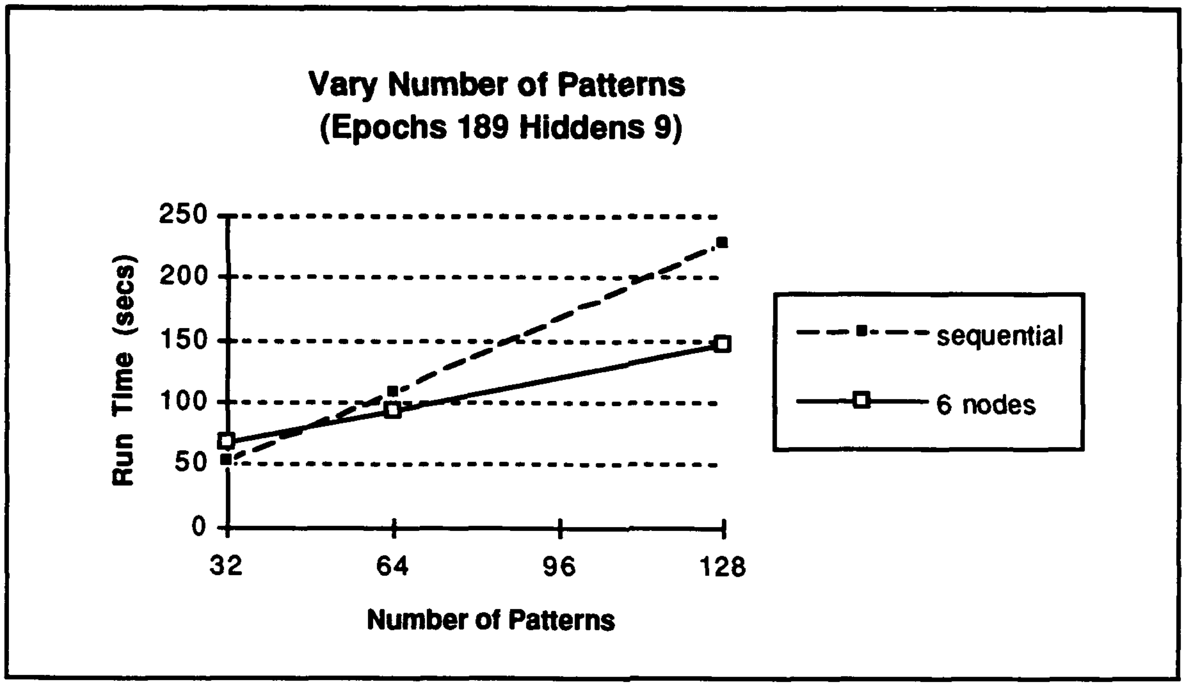

Figure 40. Vary Number of Patterns

Figure 41 shows the results of the SaM and sequential executions when the number of hidden units is varied. As can be seen in the figure, the SaM execution performs better than the sequential execution for six and nine hidden units. Further, the run-time of the SaM execution is increasing linearly as the number of hidden units used increases but at a slower rate than the sequential execution (0.4 times the sequential rate of increase when printing is enabled in SaM but not in the sequential version). This is not surprising, since increasing the number of hidden units increases the amount of computation that must be performed per epoch.

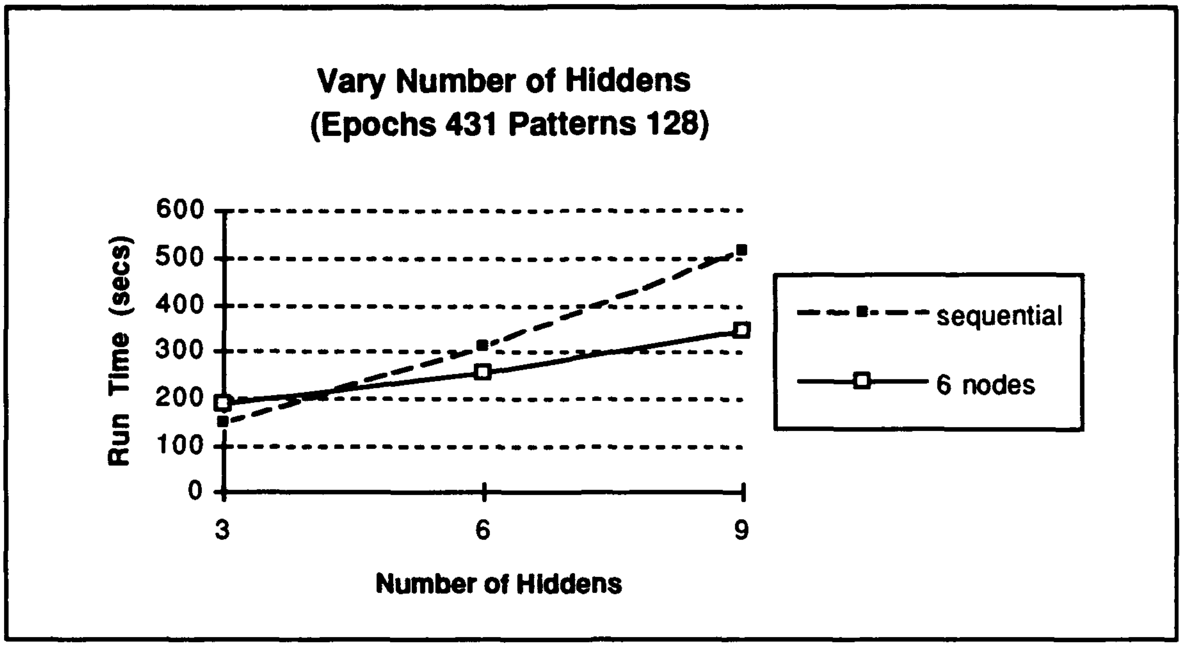

Figure 41. Vary Number of Hidden Units

Figure 42 shows the results of the SaM and sequential executions when the number of epochs are varied for the seven-bit symmetry case. As can be seen in the figure, the SaM execution performs better than the sequential execution as the number epochs increases. Further, the run-time of the SaM execution is increasing linearly as the number of epochs increases but at a slower rate than the sequential execution (0.7 times the sequential rate of increase when printing is enabled in SaM but not in the sequential version). This result was most gratifying because it suggests the viability of the SaM approach given the proper granularity of the computation and sufficient memory available on each multicomputer node.

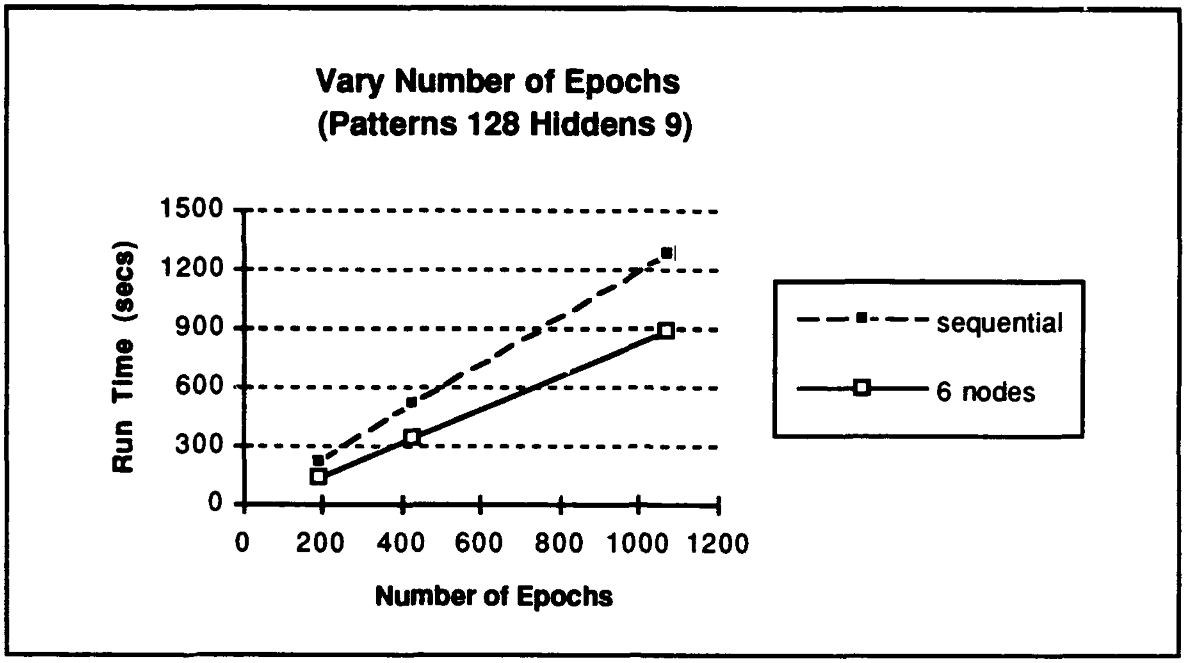

Figure 42. Vary Number of Epochs

SECTION 7

TRACKER APPLICATION

We implemented a concurrent version of a multitarget tracking application we obtained from the California Institute of Technology. The original tracker code, which is written in C, is available by anonymous ftp from ccsf.caltech.edu and can be found in the citlib/code/benchmarks directory. A more detailed discussion of the tracking algorithms embodied by this code can be found in [Gottschalk:88].

The tracker application's input consists of a series of sensor report datasets. Each dataset is a collection of (x,z) coordinates that identify the locations of targets as seen by a sensor on its 2-dimensional focal plane. Sensor reports are generated from an input file that specifies the number and locations of launch sites and, for each site, the number and launch parameters of boosters originating at the site. The application uses a simple powered flight model to generate sensor reports every five seconds of simulated time. As each new set of sensor reports is available, the application performs the following:

1. Previously recognized tracks are *extended* by identifying new sensor reports that fall within a given range of the next predicted position along each track. When more than one sensor report is sufficiently close to the predicted position, a track is *split* into multiple tracks, each ending at a different sensor report.

2. New tracks are *initiated* by looking at all possible combinations of current sensor reports that are not associated with established tracks and sensor reports from the previous two datasets.

When a new track is initiated or an existing track is split, a new entry is placed in the *track file*, which is an array of track records, each 340 bytes in length. The performance of the algorithm employed by the tracker application with respect to accuracy is documented in [Gottschalk:88]; we are only concerned here with the performance of our concurrent implementation with respect to execution time.

Since individual tracks can be processed independently from each other, we decided to create a number of tracker objects, each with its own track file and each responsible for processing a subset of all of the tracks recognized at any point during an execution. In all of our concurrent tracker implementations, the start object generates and sends each set of sensor reports to each of *n* tracker objects (see figure 43).

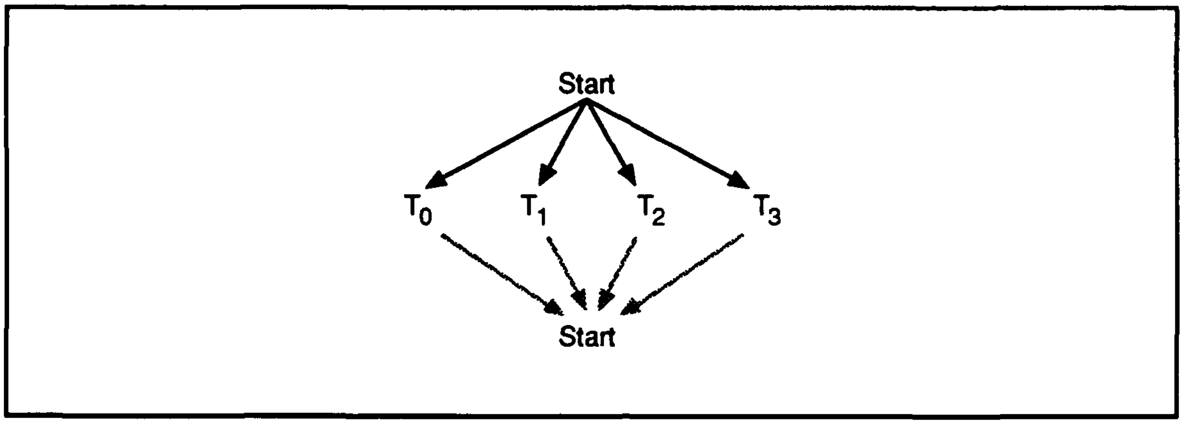

Figure 43. Initial Tracker Application Control Flow

In all of our implementations, tracker object *i* is responsible for all of the recognized tracks that currently end on a sensor report whose index in the sensor report array is *i* modulo the number of tracker objects. For the purpose of initiating new tracks, each tracker object saves two complete sets of previous sensor reports. While initiating new tracks, however, tracker object *i* only considers sensor reports whose index in the current sensor report array is *i* mod *n*. Thus, new tracks are initiated on the tracker objects that are responsible for them.

For the purpose of extending and splitting existing tracks, each tracker object must consider all of the sensor reports in the current dataset. Thus, the responsibility for a given track that ended on sensor report *i* in the previous dataset moves from tracker object *i* to tracker object *j* when the track is extended or split so that it ends on sensor report *j* in the current dataset. In this case, the track record for this track must be transferred to tracker object *j* before the next set of sensor reports is processed.

The significant difference between each of our implementations of the tracker application is how this transfer of track records is accomplished. Our initial implementation was the simplest and most straightforward. Subsequent implementations were necessary because of the limited memory available on each node of the Symult S2010 on which we ran our applications.

In our initial implementation (figure 43), each tracker object replies to the message from the start object that contains a new sensor report dataset. Each tracker object uses the data it receives from the start object to extend, split, and initiate tracks. Then it replies to the start object with an array of track records for tracks that are another tracker object's responsibility for the next simulation step. The start object has a relocation track file for each tracker object and, when it receives each reply, it copies each track record in the reply to the relocation track file for the tracker object that is responsible for the track. When all replies have been received, the start object generates the next set of sensor reports and sends the new dataset to each tracker object, along with the relocation track file for that tracker object. Before processing the new dataset, each tracker adds the track records it receives from the start object to its track file.

Our original implementation has a simple, straightforward design that minimizes the number of messages that are sent between objects and incorporates a natural mechanism for keeping the start object from racing ahead of the tracker objects. Without such a mechanism, the start object would generate all of the datasets for the entire execution, flooding the network with messages for tracker objects and opening the door for large numbers of rollbacks without some mechanism in place to keep tracker objects from proceeding to a new simulation step before all track relocations were complete for the current step.

By creating one fewer tracker objects than the number of Symult S2010 nodes running the application, each node hosts one application object (the remaining node hosts the start object). This accomplishes two things: it maximizes the granularity of the application, and all application objects can be processing messages concurrently.

The main disadvantage of our first implementation is that the start object is sufficiently large and requires sufficiently many states to be saved that it runs out of memory, even though it is the only application object on its S2010 node. The start object's size is so large because of its relocation track files, one for each tracker object and each large enough to hold the greatest number of track records that are relocated to any tracker object during an execution. For example, with 8 tracker objects and 145 threats, the greatest number of track records relocated to any tracker object is 38. In this case, the start object must maintain 8 relocation track arrays, each capable of holding 38, 340-byte track records. This adds over 100 KB to the start object's state, more than doubling its size. At the same time, the fact that the start object receives messages from all tracker objects means that additional copies of its enlarged state are saved. The limited, 8-MB memory of a S2010 node is insufficient to support this implementation with 8 tracker objects and large enough track files to handle even 45 threats.

Our second implementation was designed to reduce the size of, and the number of messages received by, the start object. In this implementation, each tracker object sends the track records that must be relocated to a new relocator object ("R" in figure 44). The relocator's entire state consists of one relocation track file for each tracker object and a counter the relocator uses to keep track of whether it has heard from all tracker objects during the current simulation step. Because the relocator must know when it has received all of the tracks that must be redistributed for the next simulation step, every tracker object must send it a message, even if the track object has no track records to give it.

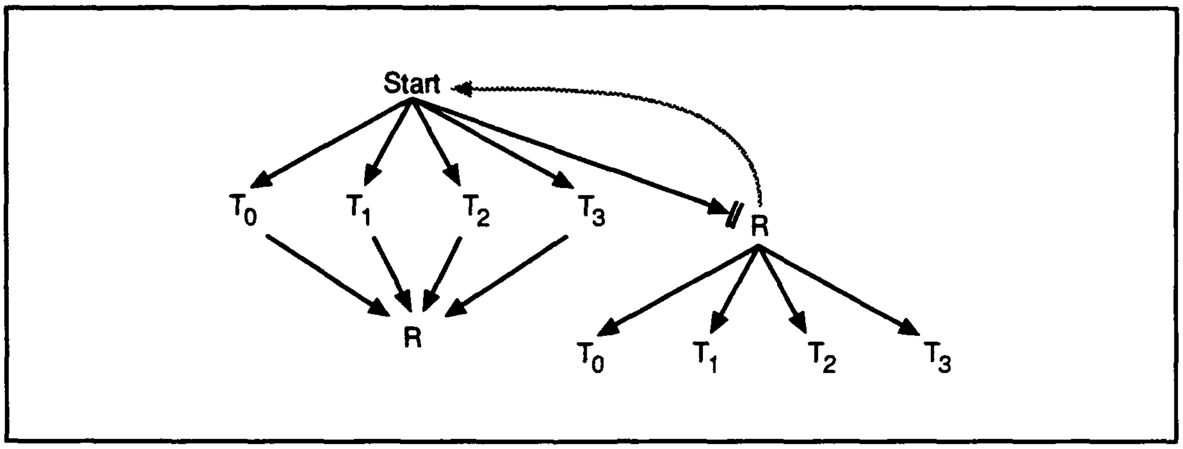

Figure 44. Tracker Application Control Flow With Relocator Object

After sending a new sensor report dataset to each tracker object, the start object sends a *redistribute* message to the relocator. The relocator's redistribute method has an assert that keeps the relocator from processing the redistribute message until its counter instance variable indicates that it has received all of the track records to be redistributed for the next simulation step. Once the relocator has heard from all tracker objects (including those with no track records to give it), it processes the redistribute message. This causes it to send the contents of its relocation track files to the corresponding tracker objects and reply to the start object. When the start object receives this reply, it begins the next simulation step.

This implementation moves the relocation track files and the messages sent to their location by the tracker objects from the start object to the relocator object. We found that this was still not enough to enable us to run with 8 tracker objects and only 45 threats. This time, the S2010 node on which the relocator object is the sole application object runs out of memory. The obvious solution is to separate the relocation track files into one object that accumulates tracks for each tracker object (see figure 45).

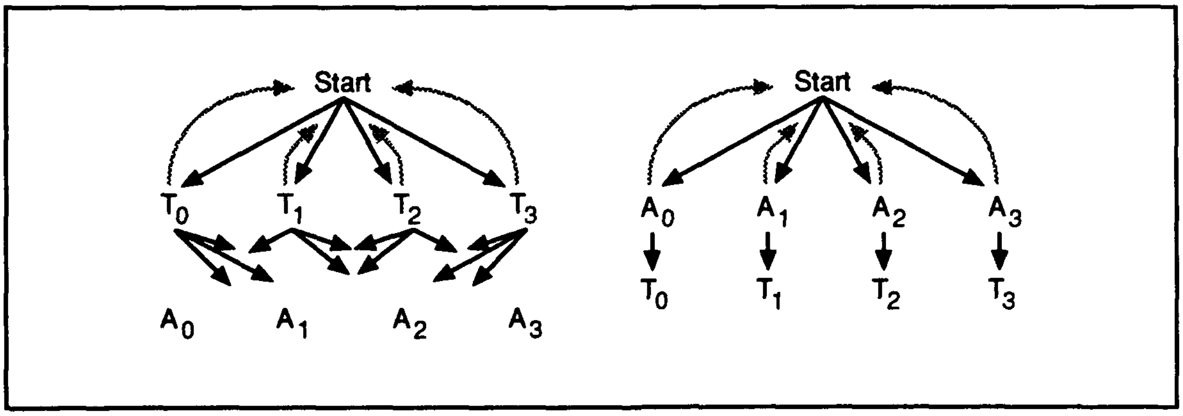

Figure 45. Tracker Application Control Flow With Accumulator Objects

In this implementation, each tracker object has a corresponding accumulator object, and each tracker object sends track records directly to the appropriate accumulator objects. It is natural to ask at this point, if not previously, why tracker objects do not send track records directly to other tracker objects. The reason is that tracker objects are by far the largest objects. This means that if memory is tight, they should receive as few messages as possible, so that as few states as possible are saved. Accumulator objects, on the other hand, are relatively small objects, since each needs only a single relocation track file. While the number of messages that each accumulator may receive during a simulation step increases in proportion to the number of tracker objects, the required size of each accumulator's relocation track file decreases accordingly. Thus, this is a more scalable solution to the track relocation problem.

The start object waits for a reply from every tracker object before sending a redistribute message to all accumulator objects. When an accumulator object receives a redistribute message, it sends the contents of its relocation track file to its tracker object. Then it empties its relocation track file in preparation for the next simulation step and sends a reply to the start object. The start object waits for a reply from every accumulator object before beginning the next simulation step. Although the relocation track files have been removed from the start object in this implementation, the start object once again receives too many messages and its S2010 node runs out of memory.

This problem could be resolved by requiring every tracker object to send a message to every accumulator, even if the tracker has no track records for the accumulator. This would permit a solution similar to the previous implementation with an assert defined for the accumulator's redistribute method. In this case, the start object would not need to receive replies from tracker objects; it would still need to receive a reply from every accumulator, however.

The way we solved this problem in our final implementation was by inserting a controller object ("C" in figure 46) between the start object and the other objects in the application. The controller object has no instance variables; it only serves as a fanout for messages from the start object and a fanin for replies. We can run our largest scenario, which consists of 13 tracker objects and 145 threats, using this implementation. With 13 trackers, 13 accumulators, one controller, and one start object, we can place one application object on each of the 28 nodes in our Symult S2010. The 145 threats represent essentially the entire threat data file obtained from Caltech. (We tried running with additional trackers by locating corresponding trackers and accumulators on the same node, which might be desirable if memory were not an issue. Unfortunately, the S2010's limited memory cannot support this mode of operation.)

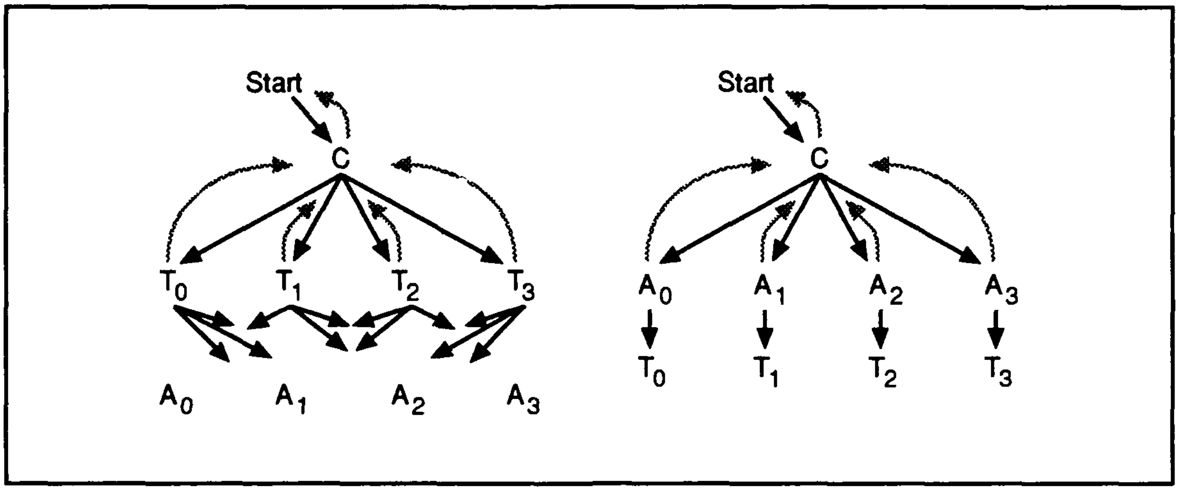

Figure 46. Final Tracker Application Control Flow

We ran our final implementation in three configurations: 13 trackers, 8 trackers, and sequential. The 13-tracker configuration ran on 28 Symult nodes, placing one application object (13 trackers, 13 accumulators, 1 start object, and 1 controller) on each node and using SaM for synchronization. Data was gathered from 13-tracker executions with threat scenarios consisting of 25, 45, 65, 85, 105, 125, and 145 threats. The 8-tracker configuration ran on 18 Symult nodes, again placing one application object on each node and using SaM for synchronization. Data was gathered from 8-tracker executions with threat scenarios consisting of 25, 45, 65, 85, and 105 threats. The sequential configuration ran on 1 Symult node and did not use SaM for synchronization. Data was gathered from sequential executions with threat scenarios consisting of 25, 45, 65, 85, 105, 125, and 145 threats.

The number of threats that we can process with our concurrent implementation is limited by the memory size of our Symult nodes and the rate at which GVT computation can be performed. The largest objects in our final implementation are the tracker objects. A tracker's instance variables include over 90 KB of tables, filters, and other data that is used to store and process sensor report datasets. In addition to these instance variables there is the track file itself, which must be large enough to store as many 340-byte track records as necessary at any point during an execution.

The minimum track file size depends on the number of tracker objects as well as the number of threats. With our 145-threat scenario and 13 trackers, track files must be large enough to store 99 track records. (Even though each tracker eventually ends up with only 11 or 12 track records in its track file, there are as many as 99 *provisional* tracks that must be stored in a single tracker's track file before they become disambiguated.) This makes the minimum size of tracker objects approximately 131 KB in a 13-tracker configuration capable of processing up to 145 threats. With only 8 trackers, our 145-threat scenario requires that track files be large enough to store 149 track records.

Thus, the minimum size of tracker objects is approximately 148 KB in an 8-tracker configuration capable of processing up to 145 threats.

This difference of 17 KB in the size of tracker objects is why we cannot run an 8-tracker configuration with 145 threats. When we do, execution proceeds 1 or 2 simulation steps and then we receive error messages from every tracker's node reporting that it has insufficient memory to create another copy of its tracker object's state.

We also received these messages when we first tried to run a 13-tracker configuration. We solved the problem in this case by increasing the rate at which GVT computations are performed. The gvt_master on node 0 waits at least *GVT_Interval* before beginning a new GVT computation. The value of GVT_Interval we were using was 1000 milliseconds. This is the same value that we used for all executions of the image processing and neural net applications discussed above. For our 13-tracker configuration, however, a GVT_Interval of 1000 milliseconds does not result in memory being recovered sufficiently fast to permit new tracker states to be created as quickly as necessary for the execution to continue. By decreasing GVT_Interval to 300 milliseconds, however, we can run the 13-tracker configuration with 145 threats.

When we subsequently saw the 8-tracker configuration run out of memory, we proceeded to lower the value of GVT_Interval even further. At values below 50 milliseconds, however, the algorithm we use to compute GVT fails. Recall from our earlier discussion of GVT that the 3-phase algorithm we use works properly as long as the first phase is long enough for all messages that are in transit when it begins to be delivered before it ends. This condition is not satisfied with a value of less than 50 milliseconds for GVT_Interval. At a GVT_Interval of precisely 50 milliseconds, tracker states are still too large at 148 KB. We have to reduce the maximum number of threats to 105 to produce an 8-tracker configuration that does not run out of memory. With 105 threats, the 8-tracker configuration requires track files large enough to store 122 track records, which makes the minimum size of tracker objects approximately 136 KB.

Many GVT computations are performed with GVT_Interval set to 50 milliseconds. Table 7 shows some of the interesting data gathered during an 8-tracker execution with 105 threats. We can see that 258 GVT computations were completed during this 154-second execution. Each GVT computation involves 4 sets of messages sent between the gvt_master on node 0 and the gvt_coordinators on the other 17 nodes, or 68 messages total. This means that the 253 GVT computations performed during this execution account for 17,544 of the 22,340 external messages shown in table 7. In other words, GVT computation was responsible for over 78 percent of the messages between processors during this execution.

Table 7. Data Collected During an 8-Tracker Run With 105 Threats

Application objects	18
Contexts	1,815
Futures	132
Multifutures	115
States saved	16,050
Instance variables saved	1,811
Environments saved	8,804
External messages	22,340
Internal messages	5,623
GVTs	258
Positive rollbacks	253
Negative rollbacks	118
States rolled back	801
Run time (seconds)	153.944

GVT computation accounts for most interprocessor message traffic even in 13-tracker executions with a GVT_Interval of 300 milliseconds. Table 8 shows data gathered during a 13-tracker execution with 145 threats. The 13,716 external messages required to perform 127 GVT computations account for over 65 percent of the messages between processors during this execution.

Table 8. Data Collected During a 13-Tracker Run With 145 Threats

Application objects	28
Contexts	2,774
Futures	142
Multifutures	115
States saved	24,247
Instance variables saved	2,751
Environments saved	13,186
External messages	21,009
Internal messages	7,187
GVTs	127
Positive rollbacks	455
Negative rollbacks	193
States rolled back	1,704
Run time (seconds)	184.664

This application was also subject to the effects of printing discussed above with respect to the image processing application. Figure 47 shows the run times of our 13-tracker, 8-tracker, and sequential configurations as a function of threat size. This data was taken with printing enabled. With printing enabled, the start object directs all trackers to print the contents of their track files after the final simulation step has been completed. It is not clear from this figure whether an 8- or 13-tracker configuration would perform better than a sequential execution with additional threats. Unfortunately, we cannot perform the requisite measurements on the Symult S2010.

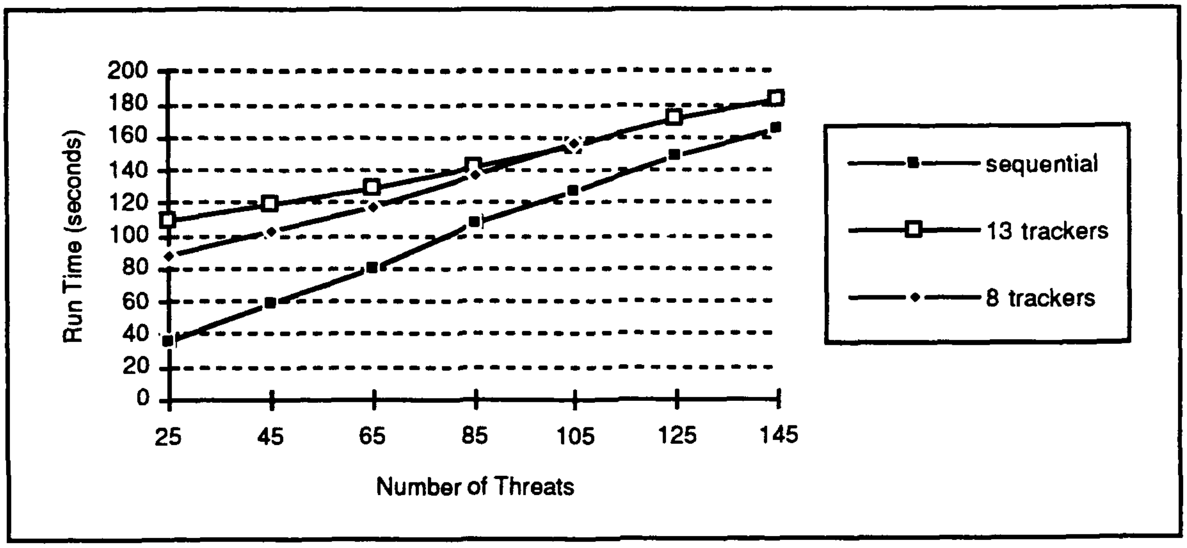

Figure 47. Tracker Application Run Times With Printing

Figure 48 shows the corresponding run times when printing is disabled. From this figure, it appears that our 8-tracker configuration would perform better than a sequential execution if it could handle more than 105 threats, our 13-tracker configuration does perform better than a sequential execution with more than 125 threats, and our 13-tracker configuration might perform better than our 8-tracker configuration if it could handle additional threats.

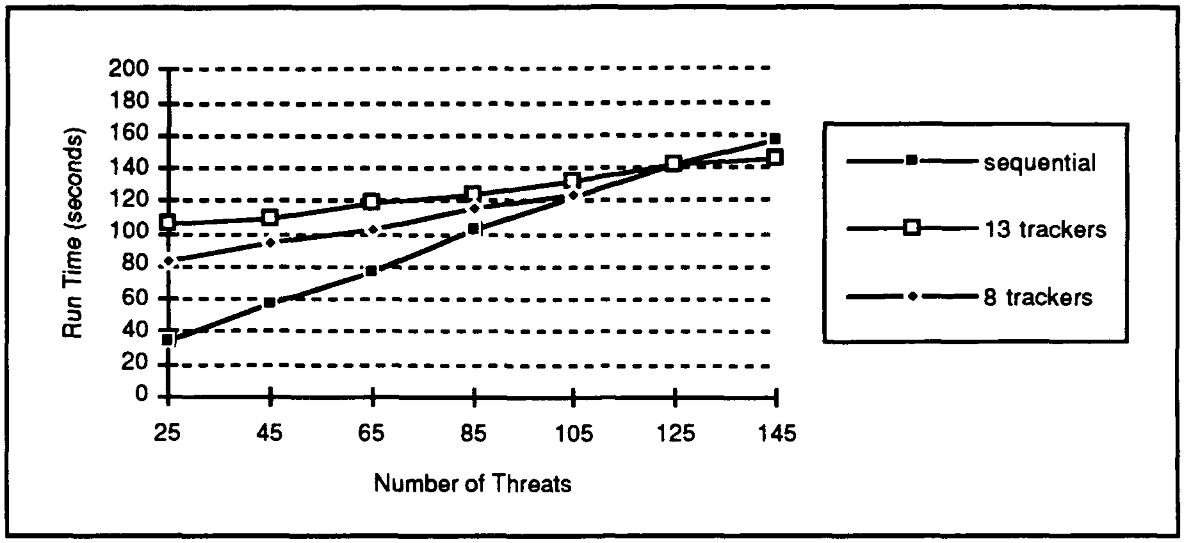

Figure 48. Tracker Application Run Times Without Printing

The nonlinear effects of printing as observed above with respect to the image processing application seem to be present here as well. Figure 49 shows the difference in run times between runs in which printing was enabled and disabled. Although the number of lines printed during tracker runs is well below the levels at which the nonlinear effects of printing become pronounced, this figure does appear to suggest a nonlinear increase in run time as the 8-tracker configuration goes from printing 89 lines (at 25 threats) to printing 169 lines (at 105 threats) and the 13-tracker configuration goes from printing 129 lines (at 25 threats) to printing 249 lines (at 145 threats).

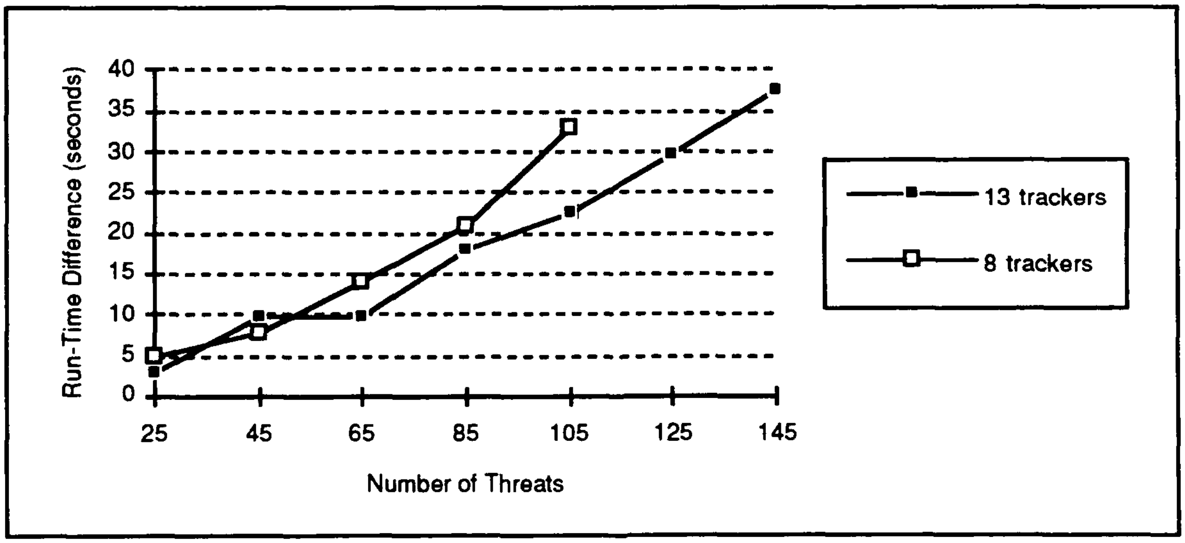

Figure 49. Run-Time Difference Between Printing and Nonprinting Runs

SECTION 8

CONCLUDING REMARKS

In this paper, we briefly described the synchronization manager as well as the enhancements made to SaM and CPM this year to improve performance and provide language constructs to allow the application program developer to control synchronization directly. Performance improvements included a new method for performing Global Virtual Time (GVT) calculations and a new method for handling send-to-self method invocations. New parallel language constructs included assert statements and statements for manipulating multifutures. We also described our experience using SaM to run several application programs written in CPM on a Symult S2010 multicomputer. The applications we developed and tested included image perspective transformation, neural network training, and multiple target tracking.

In the image processing application section, we described alternative implementations of a perspective transformation application. We observed that for an image that contains 200 vertices, the run-time of the application using SaM on four S2010 nodes performed better than the same application without SaM, run sequentially on one S2010 when more than one view is computed. Furthermore, the run-time of the SaM execution increased linearly with the number of views computed but at a slower rate than the run-time of the sequential execution (roughly half the rate). When printing was suppressed, we observed that performance generally improved when more nodes were used for the computation and that run-time increased linearly with the number of views computed. When printing was enabled on three or four nodes, the run-time also increased linearly with the number of views computed. However, the performance was generally worse when more nodes were used and run-time increased non-linearly with the number of views.

We explored various explanations for this phenomena. We showed that the output object is not a bottleneck. We showed that the actual printing itself is not responsible. We looked in detail at various parameters collected during executions of the application on four nodes and seven nodes, to try to determine if the seven-node implementation is doing more work than the four-node implementation as the number of views increases when printing is enabled. On nodes other than zero, both more application work and more SaM work is being done in the four-node executions. On node zero, although more work is being done in the seven-node implementation, the amount of work performed increases linearly with the number of views computed, except for the number of external messages sent by node zero. The cause of these additional external messages is that additional GVT computations are being performed because the program is running longer in the seven-node implementation. At this point we are unable to identify the cause of the non-linear behavior.

In the neural network application section, we described alternative implementations for training a neural network. The SaM execution is slower than the sequential execution when the training set consists of 32 patterns, but faster for 64 or 128 patterns. (These tests were run with nine hidden units for 189 epochs.) The SaM execution is slower than the sequential execution when three hidden units are used, but faster when six or nine hidden units are used. (These tests were run with 128 patterns for 431 epochs.) The SaM execution is faster than the sequential execution when the number of epochs is 189, 431, 1074. (These tests were run with 128 patterns and nine hidden units.) Further, the run-time of the SaM execution increases linearly with the number of patterns, hidden units, and epochs but at a slower rate than the sequential execution (0.4, 0.4, and 0.7 times the sequential rate of increase, respectively, when printing is enabled in SaM but not in the sequential version).

In the tracker application section, we described a series of alternative implementations for multiple target tracking that was driven by a need to reduce application memory requirements to accommodate the limited memory on Symult S2010 nodes. We were able to reduce memory requirements sufficiently in our final implementation to be able to track up to 105 targets with 8 tracker objects and up to 145 targets with 13 tracker objects. With printing enabled, neither configuration performed better than a sequential execution, and it was unclear whether either would do so at higher numbers of threats. With printing disabled, however, both of the parallel configurations performed better than a sequential execution, given a sufficient number of threats.

The lack of sufficient memory available on each S2010 node to support application program and data has severely restricted our ability to test SaM on really large versions of our applications. However, in spite of the challenges presented by the S2010 as a multicomputer platform in general and in particular as a platform to support SaM, we believe we have been able to show the correctness and viability of our basic approach.

LIST OF REFERENCES

[Agha:86] Agha, G. A., *ACTORS: A Model of Concurrent Computation in Distributed Systems*, Cambridge, MA: MIT Press, 1986.

[Ammeral:86] Ammeral, L. *Programming Principles in Computer Graphics* , John Wiley & Sons, New York, N. Y., 1986.

[Bellenot:90] Bellenot, S., "Global Virtual Time Algorithms," Proceedings of the 1990 SCS Conference on Distributed Simulation, Vol. 22, No. 2, Society for Computer Simulation, San Diego, January, 1990.

[Brando:92] T. J. Brando, and M. J. Prelle, "Fault-Tolerant Synchronization Management for concurrent Object-Oriented Computation," MTR 92B0000048, The MITRE Corporation, March 1992.

[Bridgland:91] Bridgland, M. F., J. I. Leivent, and R. J. Watro, "Mathematical Foundations for Time Warp Systems," MTR 10959, The MITRE Corporation, January 1991.

[Chatterjee:89] Chatterjee, A., "Futures: A Mechanism For Concurrency Among Objects," Proceedings Supercomputing Conference, Reno, NV, November 1989.

[Chow:87] Chow, E., H. Madan, and J. Peterson, "A Real-Time Adaptive Message Routing Network for the Hypercube Computer," Proceedings Eighth Real-Time Systems Symposium, IEEE Computer Society, December 1987.

[Gottschalk:88] Gottschalk, T. D., 1988, "Concurrent Multiple Target Tracking," Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, ACM, January 1988, Pasadena, CA, pp. 1247–1268.

[Halstead:85] Halstead, R. H., "MultiLisp: A Language for Concurrent Symbolic Computation," ACM Transactions on Programming Languages and Systems, pp. 501–538, October 1985.

[Hoare:78] Hoare, C. A. R., "Communicating Sequential Processes," *CACM* 21:8, 1978, 666-677.

[Jefferson:87] Jefferson, D., et al., "Distributed Simulation and the Time Warp Operating System," Proceedings Eleventh ACM Symposium on Operating Systems Principles, Austin, TX, November 1987.

[Liskov:88] Liskov, B. "Distributed Programming in Argus," Communications of the ACM, Vol. 31, No. 3, March 1988.

[McClelland:88] McClelland, J. L., and D. E. Rumelhart, "Explorations in Parallel Distributed Processing: A Handbook of Models, Programs, and Exercises ," Bradford Books/MIT Press, Cambridge, MA, 1988.

[Pomerleau:88] Pomerleau, D. A., G. L. Gusciora, D. S. Touretzky, and H. T. Kung, "Neural Network Simulation at Warp Speed: How We Got 17 Million Connections Per Second," Proceedings of the IEEE International Conference on Neural Networks, San Diego, CA, July 24-27 1988.

[Prelle:90] Prelle, M. J., T. J. Brando, E. H. Bensley, J. I. Leivent, R. J. Watro, and A. M. Wollrath, "Distributed Object-Oriented Programming FY90 Final Report," MTR 11058, The MITRE Corporation, December 1990.

[Prelle:91] Prelle, M. J., A. M. Wollrath, T. J. Brando, E. H. Bensley, "The Impact of Selected Concurrent Language Constructs on the SaM Runtime System," OOPS Messenger, ACM Press, Vol. 2, No. 2, April, 1991.

[Prelle:92] Prelle, M. J. and A. M. Wollrath, "The SaM Synchronization Manager—Distributed Object-Oriented Programming FY91 Final Report," MTR11229, January 1992.

[Reiher:90] Reiher, P., R. Fujimoto, S. Bellenot, and D. Jefferson, "Cancellation Strategies in Optimistic Execution Systems," Proceedings of the SCS Multiconference on Distributed Simulation, San Diego, CA, January 1990.

[Samadi:85] Samadi, B., "Distributed Simulation, Algorithms and Performance," Ph.D. dissertation, UCLA, 1985.

[Tinker:88] Tinker, P., and M. Katz, "Parallel Execution of Sequential Scheme with ParaTran," Proceedings of the Conference on Lisp and Functional Programming, Snowbird, Utah, July 1988.

[Yonezawa:87] Yonezawa, A., E. Shibayama, T. Takada, and Y. Honda, "Modeling and Programming in Object-Oriented Concurrent Language ABCL/1," in Object-Oriented Concurrent Programming, edited by A. Yonezawa and M. Tokoro, Cambridge, MA: MIT Press, 1987.

APPENDIX

APPLICATION LANGUAGE EXTENSIONS

free

Usage

`free(`*instance* `)`

Description

When an instance is no longer needed, the memory associated with that object can be deallocated. The only argument to the `free` function is the instance that was created via a `make_instance` call.

Examples

```
#include "/vb/ann/cpm/lib/cpm.h" // includes all necessary cpm utilities
                                 // including the string library class
class clock plain_clock;
class cuckoo_clock fancy_clock;
smString plain_name, fancy_name;

plain_name.init(“timex”);
fancy_name.init(“rolex”);

// making use of the default make_instance for the class clock...
plain_clock = make_instance(clock, plain_name);

// if the application programmer redefines make_instance to accept
// one or more user-defined arguments, the following is possible...
fancy_clock = make_instance(cuckoo_clock, fancy_name, NumChimes);

// some other interesting code here...

free(plain_clock);
free(fancy_clock);

// NOTE: you do not free smStrings since they were not allocated
// using make_instance (they are statically allocated).
```

Multifutures

Using multifutures gives the programmer some control over concurrency in the application program. By adding the result of a send or make_instance to a multifuture, a parallel thread of computation is created. The threads can be created first, before the result of their computation is needed, thus increasing parallelism in the application program. Using multifutures is a convenient way to reduce the overhead incurred by the synchronization manager and to avoid unnecessary synchronizations in the application program; at the same time, multifutures give the programmer some control over *when* concurrency is exploited.

make_multifuture

Usage

`make_multifuture` (*multifuture-name*)
`multifuture` *multifuture-name;*

Description

The function `make_multifuture` is a directive to the synchronization manager to create a multifuture. Any multifuture should be declared using a `multifuture` type declaration. The following manual pages will describe how to add to and retrieve values from the multifuture.

Examples

multifuture mf;
make_multifuture(mf);

add_multifuture

Usage

`add_multifuture` (*multifuture-name, argument*)

Description

Once the multifuture is created, values can be added until the `end_multifuture` function is called (indicating no more values will be sent to the multifuture). The function `add_multifuture` redirects the result of the send (which is the *argument*) to the multifuture indicated by *multifuture-name*. No limit is placed on the number of values that can be added to a multifuture (except for system memory requirements).

Examples

```
multifuture mf;                    // declare multifuture
make_multifuture(mf);              // create multifuture
add_multifuture(mf, b->get_x()); // result of b->get_x() is sent to mf
add_multifuture(mf, b->get_y()); // result of b->get_y() is sent to mf
```

end_multifuture

Usage

`end_multifuture` (*multifuture-name*)

Description

The function `end_multifuture` indicates that no more values will be added to the multifuture (since the synchronization manager needs to be aware of the number of values in the multifuture).

Examples

```
multifuture mf;
make_multifuture(mf);
for (i=0; i<NUM_FOOS; i=i+1)
        add_multifuture(mf, make_instance(foo));
end_multifuture(mf);
```

get_multifuture

Usage

`get_multifuture` (*multifuture-name, result_array, start, end*)

Description

The function `get_multifuture` retrieves values from the multifuture, *multifuture-name*. The *result_array* should be declared large enough to fit *all* the values added to the multifuture. *start* and *end* indicate the range of indices for values; the *start* parameter specifies the starting index of the values you wish to retrieve, and the *end* parameter specifies the index of the last value to be retrieved from the multifuture. When this function completes, the locations in array *result_array*[*start*] to *result_array*[*end*] will contain the values retrieved by the multifuture.

Examples

```
multifuture mf;
class foo foo_result[NUM_FOOS];

make_multifuture(mf);
// add values to multifuture...
for (i=0; i<NUM_FOOS; i=i+1)
        add_multifuture(mf, make_instance(foo));
end_multifuture(mf);
// now, retrieve values...
get_multifuture(mf, foo_result, 0, NUM_FOOS);
// alternatively, these two statements can replace the one above...
get_multifuture(mf, foo_result, 0, 5);
get_multifuture(mf, foo_result, 6, NUM_FOOS);
// now, send each foo a print message...
for (i=0; i<NUM_FOOS; i=i+1)
        foo_result[i]->print();
```

get_available_multifuture

Usage

`int get_available_multifuture` (*multifuture-name, result_array, start*)

Description

The function `get_available_multifuture` is similar to `get_multifuture` except that no ending index is specified. This function retrieves the values that the multifuture contains *so far*. The number of values retrieved is returned by `get_available_multifuture`. Note: at least one value will be retrieved.

Examples

```
multifuture mf;
class foo foo_result[NUM_FOOS];
nfuture int i, num_got, num_values;

make_multifuture(mf);
for (i=0; i<NUM_FOOS; i=i+1)
        add_multifuture(mf, make_instance(foo));
end_multifuture(mf);
num_values = 0;
while (num_values < NUM_FOOS) {
        // get the values as they come in...
        num_got = get_available_multifuture (mf, foo_result, num_values);
        for (i=num_values; i<num_got + num_values; i=i+1)
                foo_result[i]->do_useful_work();
        num_values = num_got + num_values;
}
```

free_multifuture

Usage

`free_multifuture` (*multifuture-name*)

Description

The function `free_multifuture` indicates that the multifuture is no longer needed by the computation and that memory occupied by the multifuture may be freed when the computation is committed (that is, when GVT passes the timestamp of the free message).

Examples

```
class foo foo_result[NUM_FOOS];
multifuture mf;

make_multifuture(mf);
for (i=0; i<NUM_FOOS; i=i+1)
        add_multifuture(mf, make_instance(foo));
end_multifuture(mf);
// now, retrieve values...
get_multifuture(mf, foo_result, 0, NUM_FOOS);
free_multifuture(mf);
// now, send each foo a print message...
for (i=0; i<NUM_FOOS; i=i+1)
        foo_result[i]->print();
```

touch

Usage

`touch` (*future*[,*future*]*)

Description

In some application programs, the programmer may wish to resolve a future before passing it along to other objects. Resolving a future and instead transmitting its *value* to multiple objects saves each object from resolving the future itself. Resolving futures in advance can speed up application execution, since many methods will not have to block while a future is being resolved that otherwise would have blocked had the future not been resolved by the caller. The `touch()` function resolves each *future* in the parameter list to its respective value. `touch` can take any number of arguments each of which must be a future. Any variable that is not declared with the `nfuture` declaration may be a future and can be resolved using `touch`.

Examples

```
#include "/vb/ann/cpm/lib/cpm.h" // includes all necessary cpm utilities
                                 // including the string library class
class watch timex, movado;
smString watch_name;
int hours, minutes;                 // these two can be futures

watch_name.init(“timex”);                              // make a timex
timex = make_instance(watch, watch_name);
watch_name.init(“movado”);                             // make a movado
movado = make_instance(watch, watch_name);
...
hours = timex->get_hours();                            // get time
minutes = timex->get_minutes();
touch(hours, minutes);                                 // resolve futures
movado->set_time(hours, minutes);                      // set other watch
printf(“The time is %2d:%2d\n”, hours, minutes);   // print time
```

Note: If the `touch` was not done in the above example, then both `hours` and `minutes` (both futures after being set) would have been resolved twice—once in the current method, and once in the watch::set_time() method.

set_object_type

set_object_type provides an interface that allows the user to specify the kind of synchronization manager an application object should have. A Normal synchronization manager application object is the only one supported at this time, however, it might be useful to have special synchronization managers such as Read-Only, Write-Once-Only, or Wild at some point in the future. In the current implementation of SaM, creators are associated with Read-Only synchronization managers; futures are associated with Write-Once-Only synchronization managers. Extending this idea to allow the user to identify objects seems a natural enhancement. Wild application objects would have synchronization managers that support an input queue but ignore timestamps entirely and do not save states. Wild objects would not be allowed to interact with normal objects except through a future. A normal object could create a wild object, send it a message expecting a reply. Wild objects could create other wild objects to perform a part of the computation that requires no synchronization. The result of a wild computation would be returned through the future that was created by the normal object.
In order to set the object type simply do:

```
set_object_type (READ-ONLY);
set_object_type (WILD);
```

If no type is set NORMAL is the default.

get_object_type

There is also an interface to find the object type for the next object that will be created (if not changed by set_object_type()):

```
next_type = get_object_type();
switch (next_type) {
        case NORMAL:
                printf("next object is NORMAL\n"); break;
        case READ-ONLY:
                printf("next object is READ-ONLY\n"); break;
        case WILD:
                printf("next object is WILD\n"); break;
}
```

set_object_name

set_object_name provides an interface for allowing synchronization managers to have names associated with them. This is useful as a debugging and performance enhancement device, for example, to see which objects are experiencing the most rollbacks. In order to set an application object's synchronization manager's name simply do:

```
a_name.init("Kumquat");
set_object_name (a_name);
kumquat = make_instance(cat);
```

If no name is set the empty string is the default.

set_next_creator

set_next_creator provides an interface that allows the user to determine which processor the next object is created on. In this way the user can control the placement of individual objects. For example, a copy of a read-only object (or maybe even a write-occasionally object) can be placed on every processor. Other objects that require access to that object could be told about the one that is on their processor. In order to set the next creator simply do:

```
set_next_creator (some_processor_number);
foo_obj = make_instance(foo, ...);
```

get_next_creator

There is also an interface to find the next creator that the next object will be created on (if not changed by set_next_creator()):

```
proc_num = get_next_creator();
printf("The next processor is: %d", proc_num);
```

assert

For any concurrent class message, the programmer can define an assert method which returns a boolean value indicating whether or not the message should processed, given the current values of the instance variables and arguments. The assert method may look at instance variables and arguments to the method, but it **may not** cause any futures to be resolved or send any messages to concurrent objects. The assert method need only be defined for those concurrent class methods the programmer wishes (i.e., assert methods are not required for all concurrent class methods).

This example (taken from an early neural network application) demonstrates how to use an assert method. The assert, in effect, will not allow any messages from future epochs to be processed before the message for the current epoch is processed. The programmer must be careful however when defining assert methods since an incorrectly defined assert method can prevent the program from terminating.

```
assert OutputUnit::ChangeWeights(epochCount)
        nfuture int epochCount;
{
        // Is the epochCount equal to the output unit's view of the of
        // number epochs (netEpoch)?
        return (netEpoch == epochCount);
}

void OutputUnit::ChangeWeights(epochCount)
        nfuture int epochCount;
{
        // actual text of the method

        netEpoch = netEpoch + 1;  // increment output unit's epoch count
}
```

Note that the assert method only requires the key word assert before the method (no other type is needed). An assert method also requires a return statement which returns TRUE or FALSE. Statements other than the return statement can be included in the method, but the programmer must be careful to make sure that no futures are resolved and that no messages are sent to global objects. The best policy to follow is to only reference variables that are defined with `nfuture`.